# NAVAL POSTGRADUATE SCHOOL
## Monterey, California


# THESIS

**MULTIVARIATE MOTION PLANNING OF AUTONOMOUS ROBOTS**

by

Vasilios Karamanlis

March, 1997

Thesis Advisor: Yataka Kanayama
Co-Advisor: Nelson Ludlow

**Approved for public release; distribution is unlimited.**

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY (Leave Blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | March 1997 | Master's Thesis |

**4. TITLE AND SUBTITLE**
MULTIVARIATE MOTION PLANNING OF AUTONOMOUS ROBOTS

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
Vasilios Karamanlis

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Naval Postgraduate School
Monterey, CA 93943-5000

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSORING/ MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
Approved for public release; distribution is unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

A problem of motion control in robot motion planning is to find a smooth transition while going from one path to another.

The key concept of our theory is the steering function, used to manipulate the motion of our vehicle. The steering function determines the robot's position and orientation by controlling path curvature and speed.

We also present the - neutral switching method - algorithm that provides the autonomous vehicle with the capability to determine the best leaving point which allows for a smooth transition from one path to another in a model-based polygonal world.

The above mentioned algorithm is thoroughly presented, analyzed, and programmed on a Unix workstation, and on the autonomous mobile robot Yamabico. The research data indicate that neutral switching method improved the transition results for polygon tracking, star tracking motion, and circle tracking. Moreover, neutral switching method enhances robot control and provides a more stable transition between paths than any previously known algorithm

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| Robots, autonous mobile vehicle, local motion planning, steering function, polygon tracking | 118 |
| | 16. PRICE CODE |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |

**Approved for public release; distribution is unlimited.**

# MULTIVARIATE MOTION PLANNING OF AUTONOMOUS ROBOTS

Vasilios Karamanlis
Lieutenant, Hellenic Navy
B.S., Hellenic Navy Academy, 1985

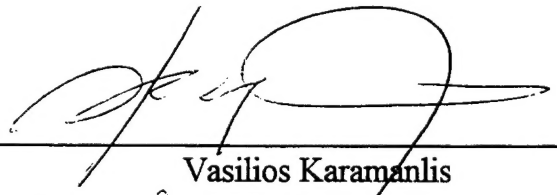
Submitted in partial fulfillment
of the requirements for the degree of

**MASTER OF SCIENCE**
**IN**
**COMPUTER SCIENCE**

from the

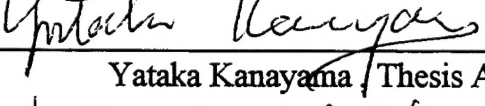
**NAVAL POSTGRADUATE SCHOOL**
**March 1997**

Author:

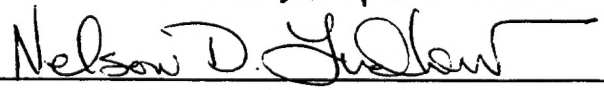


Vasilios Karamanlis

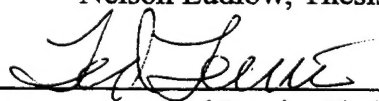
Approved by:



Yataka Kanayama, Thesis Advisor



Nelson Ludlow, Thesis Co-Advisor



Ted Lewis, Chairman
Department of Computer Science

# ABSTRACT

A problem of motion control in robot motion planning is to find a smooth transition while going from one path to another.

The key concept of our theory is the steering function, used to manipulate the motion of our vehicle. The steering function determines the robot's position and orientation by controlling path curvature and speed.

We also present the - neutral switching method - algorithm that provides the autonomous vehicle with the capability to determine the best leaving point which allows for a smooth transition from one path to another in a model-based polygonal world.

The above mentioned algorithm is thoroughly presented, analyzed, and programmed on a Unix workstation, and on the autonomous mobile robot Yamabico. The research data indicate that neutral switching method improved the transition results for polygon tracking, star tracking motion, and circle tracking. Moreover, neutral switching method enhances robot control and provides a more stable transition between paths than any previously known algorithm.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# I. INTRODUCTION

## A. BACKGROUND AND MOTIVATION

One of the most challenging problems man has ever faced and one of his ultimate goals was the creation of autonomous robots. By that we mean that the robots will be able to be programmed by high-level programming languages and will be able to solve a variety of problems from a variety of fields. Hazardous material handling, welding, painting, and assembly in factories would be some of their tasks. In addition, it will be expected that autonomous robots will be able to perform more complex jobs, such as mine searching or fire fighting. The tremendous progress in microelectronics and software industry as well as the development of related technologies gives us the ability and the power we need to build an complete autonomous robot.

The robotics and automation community is being swept by broad, pervasive technological demands. Successful deployment of automomous and reprogrammable robots is expanding the technology while theory and implementation continue to advance rapidly.

Obstacle avoidance and motion planning in general are the most common problems a robot is encountered by acting directly on the world. In addition autonomous robots are fundamentally multidisciplinary, incorporating technologies from mechanical and electrical engineering, control theory, computer vision, estimation theory, artificial intelligence, operations research, programming languages, mathematics and physics. Consequently, a vast amount of disciplines are directly or indirectly relevant to mobile-robot research.

## B. PROBLEM STATEMENT

Almost all of the above tasks depend upon our deep understanding of the motion problem. In order the robot to perform even the simplest task, a complicated combination of movements is needed. In addition it is expected to move safely in

environments filled with various objects. So as we see the expectation from a robot to move itself safely in an environment, make us deal with the problem called *motion planning problem.* Figure 1 shows an example of an environment our robot is going to perform various tasks in.

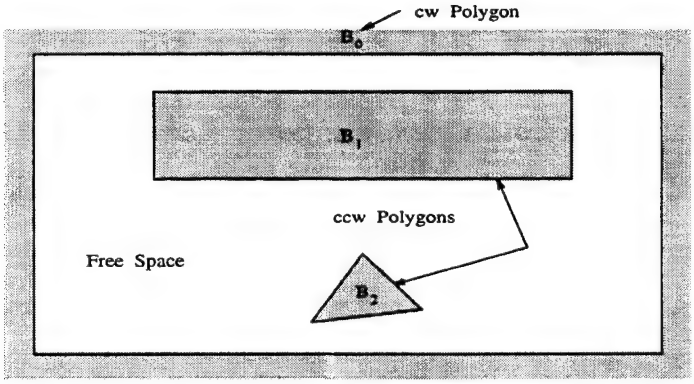



Figure 1. Robot's world space

Various techniques have been developed recently dealing with the motion planning problem and a lot of progress has been made during the last several years.

One of the most revolutionary theories is that of Professor Yataka Kanayama at NPS. This thesis concentrates on the algorithms presented in his theory which are trying to approach the motion planning problem.

## 1. Concepts Definitions and Terminology

The general motion planning problem for an autonomous vehicle can be stated as follows: Given (1) an initial state of the vehicles, (2) a desired final state of the vehicle, and (3) any constraints on allowable motions, find a collision-free motion of the vehicles from the initial state to the final state that satisfies the constraints. The above standard motion planning problem is extended and generalized in ways which give us the possibility to deal with environments which are not unknown to the robot but also dynamically changing.

Another definition of the *motion planning problem* can be given by the following schema:

"Given a robot B, an environment E, and a motion objective O, find a motion $\pi$ for B amidst E that achieves O subject to some optimality criterion C(or report that no motion exists).(Chee-Keng Yap).

Another idea that plays a major role in our research is that of *localization.* What we mean by that is the exact determination of the current position of the robot and its orientation.

One of the most fundamental concepts behind every motion planning theory is that one *local motion planning.* What we mean by that is finding the best motion among a path class using a local motion control algorithm. In our case we are going to present the steering function algorithm in details.

## 2. Problem Description

The motion planning problem is a very fundamental one in robotics-even though one of the most complex ones-and as we told before its main purpose is to enhance the robot with the capability to generate its own motion. One of the parts of this problem is called *local motion control* and we can see it in the next Figure 2.

A concept very much related to our work is that of "path classes", and after giving the necessary definitions and expanations of this concept we are going to proceed to the description of the problem we deal with in this thesis.

### *a. Path Classes*

This subsection defines a list of concepts about path classes and their importance to our problem.

A *path* $f$ in a world $\mathcal{W}$ is a continuous function

$$f : [0,1] \rightarrow free(\mathcal{W})$$

with $f(0) \neq f(1)$. We consider a path $f$ to be a directed curve with natural direction from $f(0)$ to $f(1)$. The two points $f(0)$ and $f(1)$ are called its *endpoints* and we say that the path *joins* them. We usually denote $f(0)$ as a start $S$ and $f(1)$ as a goal $G$. We assume that $f$ is rectifiable(its length is finite). A world of three *ccw* polygons

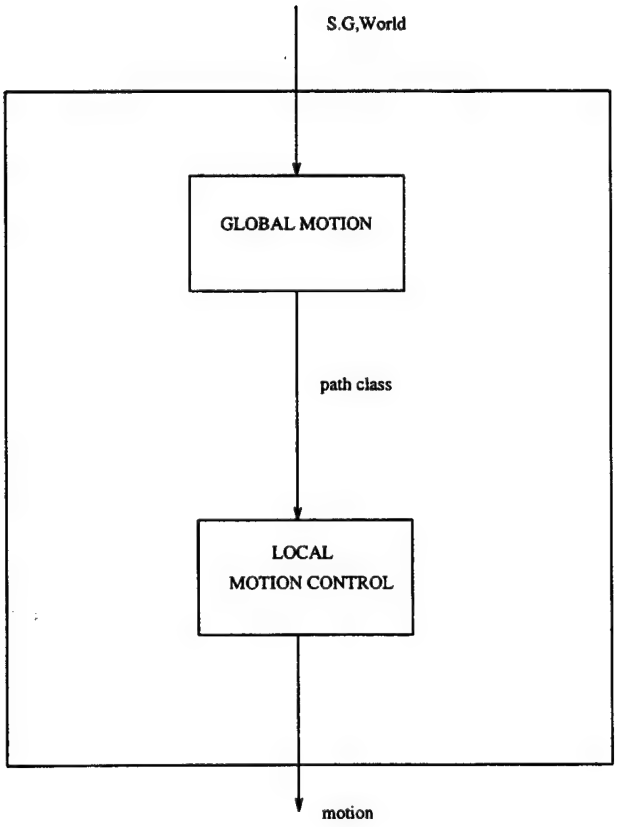



Figure 2. Motion planning problem

$B_1, B_2$ and $B_3$, one *cw* polygon $B_0$, and the paths from $S$ to $G$ can be depicted in Figure 3.

### *b. Path Classes and how we represent them*

Given a starting point $S$ and a goal point $G$ in a polygonal world $\mathcal{W}$, our effort is to find a continuous, smooth path that connects the start configuration, $S$, to the goal configuration, $G$. In Figure 3 for example, we can see three different path classes. We are using the notion of directed v-edges to represent each path class. In its most general form, a path class, $\wp$, that includes a path $f$ is symbolically represented by a *path sequence.* For instance, the path classes $f_1$ and $f_2$ in Figure 3 are represented by:

$$f_1 = [B_3/B_0]\,[B_2/B_0]\,[B_1/B_0]$$
$$f_2 = [B_3/B_0]\,[B_3/B_2]\,[B_3/B_1]\,[B_0/B_1]$$

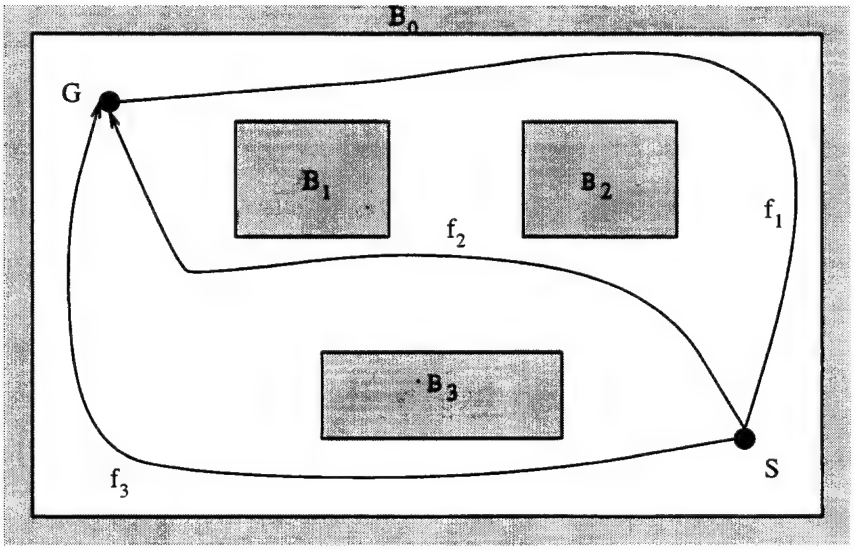



Figure 3. A world and paths

To find the best path class we have to take into consideration the "weight" of each edge related to the cost function. This best path class we've just found and the task we want our robot to execute affect the steering function and its desired motion.

As we see the objective of using path classes is to provide useful information for the local motion control problem. In our research we are going to investigate the safe navigation of an autonomous vehicle through an environment, using the steering function and the Neutral Switching method, to achieve smoothness of the motion.

## C. REVIEW OF RELATED WORK

During the last years various techniques have been used trying to solve robot-motion-planning problems. We're are going to review some of them here. Figure 3 is an example of the different paths a robot would choose to approach a given goal, and gives us the central idea of the problem we're dealing with.

We are going to say a few words about the different approaches to the motion planning problem but we're going to focus our attention more to the last previous

method used for polygon tracking. This way the reader would have a very good background in understanding the problem and he would be able to see the advantages of the new method *Neutral Switching Method* and its simplicity. There are three "classical" approaches to motion planning:*roadmap methods*,*cell decomposition methods*, and *potential field methods*.We will also describe the concept of an image and we will show how this notion was used for polygon tracking motion.

## 1. Potential Field Methods

*Potential Field Methods* can be very efficient. They are usually called *local methods* while the *roadmap* and the *cell decomposition* are called *global methods*.This method consists of defining a potential field which is represented by a function $f : \mathcal{W} \rightarrow \mathcal{R}$ and it is a combination of attractive and repulsive potentials. Attractive potentials tend to pull the robot towards the goal while at the same time repulsive potentials push the robot away from them. The negated gradient of the total potential over the free space is treated as an artificial force applied to our robot. The direction of this force shows the direction of the motion the robot should follow.

The main disadvantage of these methods is since these methods are essentially fastest descent optimization methods, the robot can become trapped in a local minima of the potential field. Some of the approaches used to avoid this problem is to design the potential functions in a way that they have no local minima, or to complement the basic potential field approach with such powerful mechanisms which give us the capability to escape from local minima.

## 2. Roadmap and Cell Decomposition methods

These two methods generally include an initial processing step aimed at capturing the connectivity of the free space in concise representation. As we mentioned above these methods are called *global methods*.

The *roadmap* method for example consists of capturing the connectivity of the robot's free space in a network of one-dimensional curves, called the *roadmap*, lying

in the free space. It is also called *skeleton approach* and how it actually works is that after a roadmap $\rho$ has been constructed the path planning is reduced to connecting the start and the goal configurations to $\rho$, and searching $\rho$ for a path.

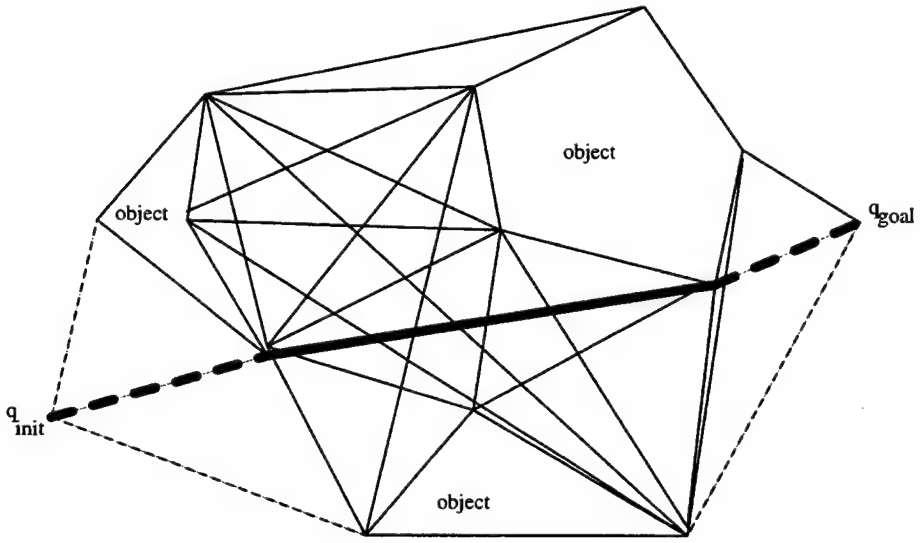



Figure 4. Visibility graph

Many other methods have been proposed based on this general idea. They include *visibility graph*, *Voronoi diagram*, *freeway net*, *silhouette* and *retraction*.

Cell decomposition methods are the most widely studied and implemented ones. The principle that lies behind them is that of decomposing the robot's free space into simple regions, called *cells*. The path planning is then performed by finding a path in $G$ from the node corresponding to the start cell (the cell containing the start configuration) to the node corresponding to the goal cell (the cell containing the goal configuration).

Both methods consist of constructing a global data structure that can later be used for solving one or more motion planning problems. There are two serious problems though:

1. The computations of the data structures tend to be very expensive in both time and memory, and

2. They do not seem to be suitable for robots with non-holonomic constraints such as car-like robots or multi-body mobile robots.

Our readers will be able to find a thorough discussion of these approaches in [9].

## 3. Polygon Tracking using Images

Given an edge L and a point p, the image, im(p,L), of p on L is defined as the closest point on L from p. The distance d(p,L) from p to L is defined as d(p,L)=d(p,im(p,L)). We assume that in our world two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ are given. The Euclidean distance $d(p_1, p_2)$ between them is defined as:

$$d(p_1, p_2) \equiv \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{I.1}$$

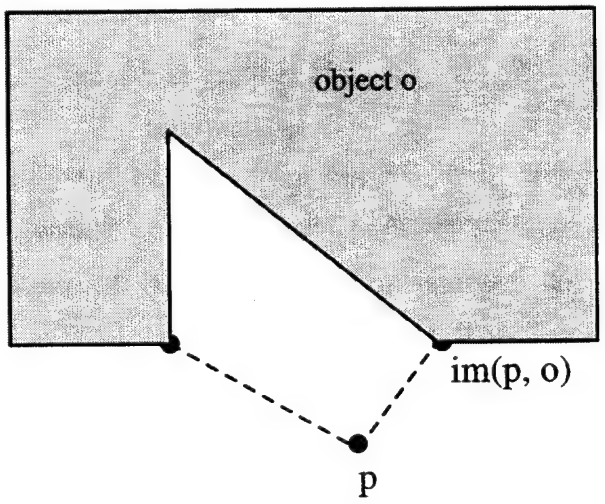



Figure 5. Image on object

Assume now that there is an object $o$ in a plane. An object might be the form of a point, a line, an open line segment, a polygon, or other set of points. We define the shortest distance $d(p, o)$ between a point $p$ and an object $o$ as follows:

$$d(p, o) \equiv \min_{p_1 \in o} d(p, p_1) \tag{I.2}$$

Eq. I.2 generalizes the function $d$ defined by Eq. I.1.

*Definition*: An image im(p,B)of a point $p \in$ free(B)on a polygon B is the closest point from p on B.

The image is a vertex on B or a point on an open edge e in B. See (Figure 5).

If a world $\mathcal{W}$ has more than one object, an image $im(p, \mathcal{W})$ is defined as the image $im(p, o_i)$ such that $d(p, o_i)$ is the minimum over all objects in $\mathcal{W}$ (Figure 6).

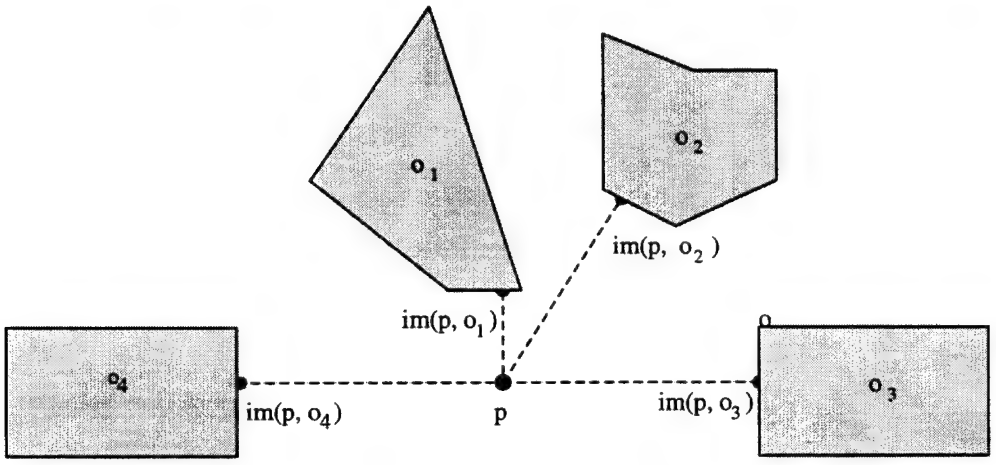



Figure 6. Images on world

Given the images in a world we have to solve the following problem: given a point $p$ in free space and a convex polygon $B$, determine whether the image from $p$ to $B$ is on an edge or on a vertex of $B$. The following definitions are useful in understanding the concepts used in this method for polygon tracking:

Assume now that we are given a convex polygon $B = (v_1, \cdots, v_n)$ and a point $p \in$ free$(B)$. The significant notion for our purpose is the following classification of each vertex $v_i$ of $B$ with respect to the segment $\overline{pv_i}$. Each vertex of $B$ is said to be *visible*, *invisible*, *cw-tangential*, or *ccw-tangential* (we should add with respect to segment $\overline{pv_i}$, but we shall normally imply this qualification) (see Figure 7).

*Definition*: Let $B$ be a convex polygon, and let a point $p \in$ free$(B)$.

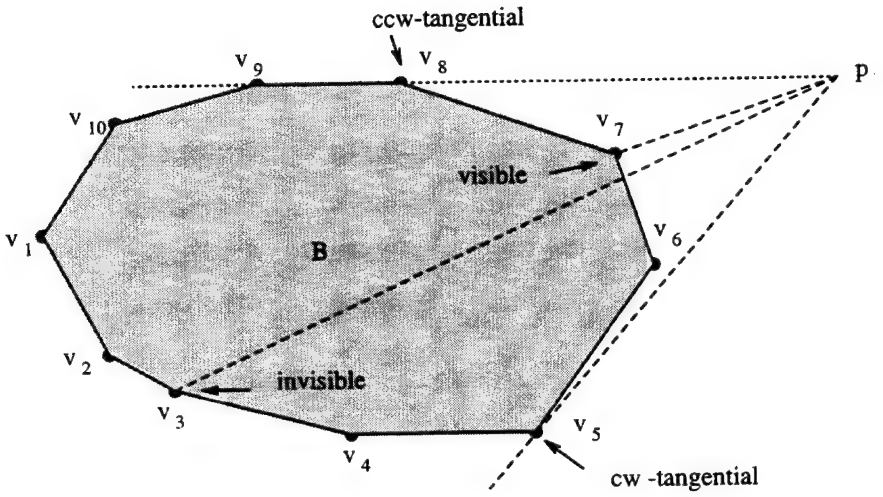



Figure 7. Visibility from point $p$ to convex polygon $B$ (I)

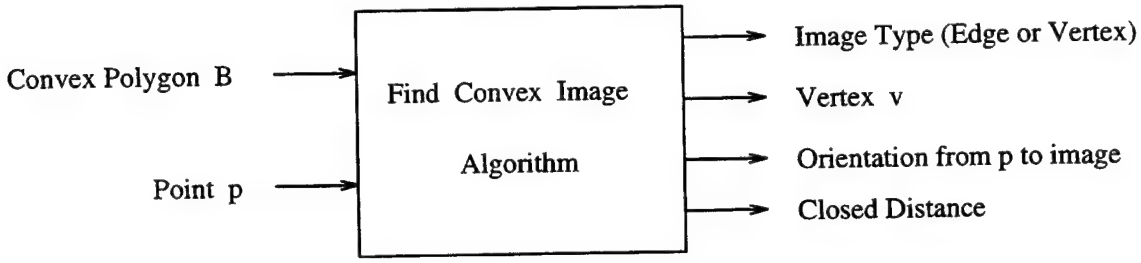



Figure 8. Image type

- A vertex $v_i$ is *tangential* from point $p$ if the two vertices adjacent to $v_i$ lie on the same side of the line containing $\overline{pv_i}$.
- A vertex $v_i$ is *visible* if the segment $\overline{pv_i}$ does not intersect the interior of $B$ and the two vertices adjacent to $v_i$ lie on opposite sides of the line containing $\overline{pv_i}$.
- A vertex $v_i$ is *invisible* if the segment $\overline{pv_i}$ intersects the interior of $B$.

# D. ASSUMPTIONS AND CONSTRAINTS

The problem of finding the optimal path for the continuous motion of a given vehicle from a given initial configuration to a desired final goal, is definitely subject to certain geometric constraints during its motion. These constraints do not permit the body of the vehicle to come in contact with certain obstacles or 'walls', of the given environment. This way issues related to the mechanical interaction are avoided.

Also our robot has a fixed number of degrees of freedom. As we see in the next figure 9 our vehicle has three degrees of freedom when moving on a flat surface. Precisely what we mean by this is the following : relative to some global coordinate system, the robot can be at any position specified by two coordinates, x and y, and pointed in any direction specified by a third coordinate angle $\theta$. These three degrees of freedom (x,y,$\theta$) give us the distance to and the angle between the global frame, and a local reference frame, R, on the robot. (We could have put the frame anywhere on the robot but because the robot's center of rotation, is the point midway between its two drive wheels, we chose that point).

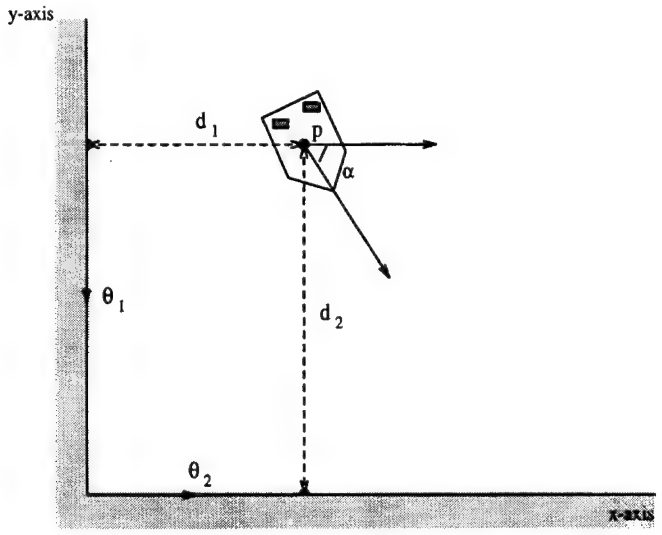



Figure 9. Representation of a robot in a global coordinate system

Also in the theory of Professor Kanayama we avoid any mathematical methods which are coordinate system-sensitive. For instance, a curve is never represented in a form of y=f(x). Most important of all according to this theory paths, motions and environments of a robot are treated as continuous entities rather than discrete ones. This policy causes some problems sometimes but make real applications easier.

Also, we believe that moving objects in general are handled better if direction is attached. So all the geometrical objects like lines, circles and curves have *directions*

(forward or backward). We believe also that this approach allows a deeper study of the inherent mathematical structure of the different problems.

# II. PATH TRACKING METHOD

## A. THE CONCEPT OF THE STEERING FUNCTION

In this chapter we introduce the mathematical framework that is used in our theory and in particular we're going to discuss the *steering function* in details. We're going to show how this function is used to manipulate the motion of our vehicle, how it is related to the robot's position and orientation and how it affects the movement of our robot in general. It is also necessary at this point, to give to the reader some important definitions we are using through out the proposed theory of Professor Kanayama.

## B. TRANSFORMATIONS

Let $\mathcal{R}$ denote the set of all real numbers.

*Definition*: A *transformation*, $q$, is defined by

$$q \equiv \begin{pmatrix} x \\ y \\ \theta \end{pmatrix},$$

where $x, y, \theta \in \mathcal{R}$.

The set of all transformations is denoted by $\mathcal{T}$. For example, $(3, 1, \pi/3)^T \in \mathcal{T}$. Obviously, a transformation $q$ is interpreted as a two dimensional coordinate transformation from the global Cartesian coordinate system $\mathcal{F}_0$ to another coordinate system $\mathcal{F}$.

*Definition*: The transformation group $\langle \mathcal{T}, \circ \rangle$ consists of the set $\mathcal{T}$ of transformations, where

$$\mathcal{T} = \{(x, y, \theta)^T | x, y, \theta \in \mathcal{R}\}$$

and the binary operator (*composition function*), $\circ$, is defined as follows:

Let $q_1 = (x_1, y_1, \theta_1)^T$, $q_2 = (x_2, y_2, \theta_2)^T \in \langle \mathcal{T}, \circ \rangle$, then

$$q_1 \circ q_2 \equiv \begin{pmatrix} x_1 + x_2 \cos\theta_1 - y_2 \sin\theta_1 \\ y_1 + x_2 \sin\theta_1 + y_2 \cos\theta_1 \\ \theta_1 + \theta_2 \end{pmatrix}.$$

The interpretation of $q_1 \circ q_2$ in the domain of two-dimensional coordinate transformations, is the *composition* of the coordinate transformations $q_1$ and $q_2$.

*Definition*: The inverse $q^{-1}$ of a given transformation $q = (x, y, \theta)^T$ is defined as:

$$q^{-1} = \begin{pmatrix} -x\cos\theta - y\sin\theta \\ x\sin\theta - y\cos\theta \\ -\theta \end{pmatrix}.$$

Another poerful tool in this transformation theory is the concept of *relative transformations*. We already know that a vehicle's configuration is represented by

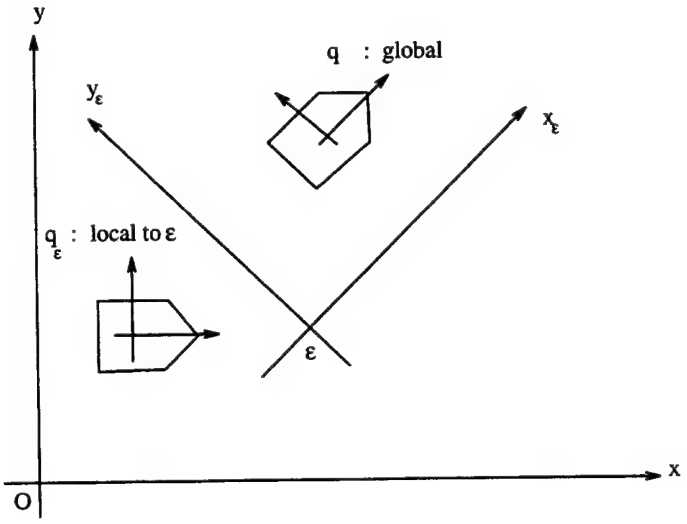



Figure 10. Robot's relative transformation

$$q_{\mathrm{V}} = (x_{\mathrm{V}}, y_{\mathrm{V}}, \theta_{\mathrm{V}})^T \tag{II.1}$$

This transformation represents the three degrees of freedom a rigid body possesses in a place.

The relative transformation now is defined as

$$q_{\mathrm{V}} = (x^\star, y^\star, \theta^\star)^T \tag{II.2}$$

of the object with respect to the vehicle 's coordinate system.There exists also a relation $q_{\mathrm{V}} \circ q^\star = q_{\mathrm{o}}$ between $q_{\mathrm{V}}$,$q_{\mathrm{o}}$ and $q^\star$.

**Proposition II.1** *If the transformations of a vehicle and an object are*

$$q_{\mathrm{V}} = (x_{\mathrm{V}}, y_{\mathrm{V}}, \theta_{\mathrm{V}})^T \text{ and } q_{\mathrm{o}} = (x_{\mathrm{o}}, y_{\mathrm{o}}, \theta_{\mathrm{o}})^T \tag{II.3}$$

*the relative transformation $q^\star$ of the object with respect to the vehicle is*

$$q^\star \equiv \begin{pmatrix} x^\star \\ y^\star \\ \theta^\star \end{pmatrix} \equiv \begin{pmatrix} (x_{\mathrm{o}} - x_{\mathrm{V}})\cos\theta_{\mathrm{V}} + (y_{\mathrm{o}} - y_{\mathrm{V}})\sin\theta_{\mathrm{V}} \\ (-x_{\mathrm{o}} - x_{\mathrm{V}})\sin\theta_{\mathrm{V}} + (y_{\mathrm{o}} - y_{\mathrm{V}})\cos\theta_{\mathrm{V}} \\ \theta_{\mathrm{o}} - \theta_{\mathrm{V}} \end{pmatrix}.$$

## 1. Steering Function Method

We know already that an ordinary vehicle has two control variables:curvature $\kappa$ and speed $v$. The assumption we make in our theory is that each vehicle operates at a relatively low speed and that speed is proportinal to the curvature k. We also take into consideration that our robot's rotational rate $\omega$ is proportional to its path curvature k if its speed $v$ is constant.

$$\omega = \kappa\, v \tag{II.4}$$

One of the assumptions we made in our introduction was that a vehicle's heading direction and curvature must be continuous. So we have to find a way to control the curvature $\kappa$ of our robot.

In order to continuously change k we should compute $\frac{d\kappa}{ds}$. Therefore we introduce and adopt through our whole study the general form of the steering function which we have tested and it works perfectly.

$$\frac{d\kappa}{ds} = -(a\Delta\kappa + b\Delta\theta + c\Delta d) \quad \text{(II.5)}$$
$$\equiv -(a(\kappa - \kappa_d) + b(\theta - \theta_d) + c\Delta d),$$

where $a$,$b$, and $c$ are positive constants, and $\Delta d$ is the "signed" distance between the vehicle and a directed line. Also, $\kappa$ is the path curvature, $\theta$ the vehicle's heading(which is equal to the path tangential direction), $\kappa_d$ the desired curvature, and $\theta_d$ the desired heading direction. $\Delta d$ is positive ($\Delta d > 0$) or negative ($\Delta d < 0$)if the robot is on the left or the right side of the reference path respectively.

A very important factor that influences the efficiency of the steering function formula is the correct choosing of the three constants ($a$, $b$, $c$). The proper tracking of our reference path depends heavily on the correct combination of these three constants.

Professor Kanayama in his theory had linearized the system for computing these constants and after solving the equation

$$\lambda^3 + a\lambda^2 + B\lambda + c = 0 \quad \text{(II.6)}$$

he found the eigenvalues $\lambda$ that satisfy that equation.

To obtain the proper condition on the coefficients for achieving asymptotic stability,Professor Kanayama had to refer to Routh-Hurwitz criterion which says that $a$, $b$, and $c$ must be all positive and $ab - c > 0$. According to this we have

$$\alpha = \kappa_1 + \kappa_2 + \kappa_3 \quad \text{(II.7)}$$
$$\beta = \kappa_2\kappa_3 + \kappa_3\kappa_1 + \kappa_1\kappa_2 \quad \text{(II.8)}$$
$$c = \kappa_1\kappa_2\kappa_3 \quad \text{(II.9)}$$

and taking all the negative real eigenvalues equal,

$$\alpha = 3\kappa \quad \text{(II.10)}$$
$$\beta = 3\kappa^2 \quad \text{(II.11)}$$
$$c = \kappa^3 \quad \text{(II.12)}$$

for some positive constant $k$.

At this point we have to say a few words about smoothness $\sigma$ and the important role it plays in the steering function.

Smoothness is very essential for our robot's navigation and motion planning in general because an unproper value of it could cause slippage of the wheels, or could cause a very unsafe motion for the vehicle .

We define

$$\sigma = \frac{1}{k}, \tag{II.13}$$

where $\kappa$ is the curvature.

In general large smoothness $\sigma$ results in faster motion and smaller smoothness in a slower one. Choosing a value for smoothness $\sigma$ depends on the environment our vehicle is going to move in. In an environment with many obstacles we need to use a smaller smoothness in order to avoid the obstacles while in the case we want to move faster we have to use larger smoothness. The next figure gives us an idea of how smoothness affects line tracing simulation results(using the old method).

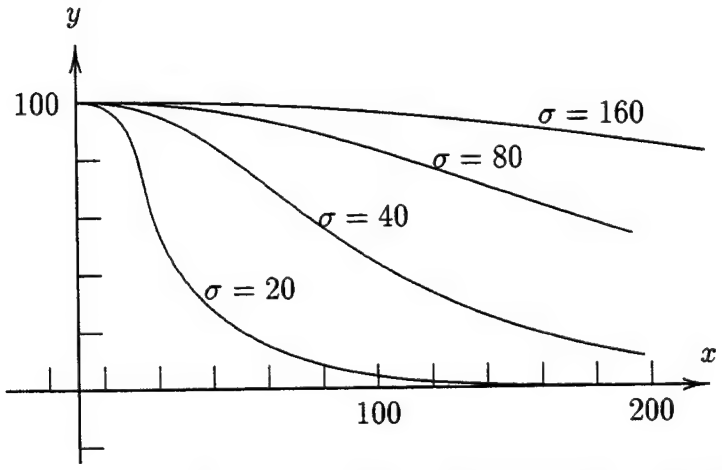



Figure 11. Effect of smoothness in line tracking

## C. NEUTRAL SWITCHING METHOD

The revolutionary method of *neutral switching* proposed by Prof.Kanayama is explained in this section. We will also show here our simulation results obtained by

working on Sun/Unix workstation in the NPS Artificial Intelligence Lab.

To ensure that our robot has smooth motion behavior, while transitioning from one path to another we have to find the best point the robot has to leave its path it currently is moving on, and proceed to the next path. So the key notion behind Neutral Switcing Method is selecting an appropriate leaving point from the current path segment our vehicle is moving on to the next one. The method we 're using gives us a unique point for each path combination.

An early leaving from the current path would have as a result an intersection with the next path something we really want to avoid while a late leaving from the current path would have as a result a strange "turning back" behavior away from the next path. Both cases are undesirable and the way to avoid them is to select a point where the steering function returns a "zero feedback" value:

$$\frac{d\kappa}{ds} = -(a\Delta\kappa + b\Delta\theta + c\Delta d) = 0$$

For the beginning we apply this principle to a line-to-line switching and more specifically from an Y axis to the X axis. When the vehicle is traveling on the Y axis down, its curvature is 0 and its orientation is equal to $-\frac{\pi}{2}$. So we obtain

$$\frac{d\kappa}{ds} = -(a\Delta\kappa + b\Delta\theta + c\Delta d) = \left(b(-\frac{\pi}{2}) + cy\right) = 0$$

Therefore

$$y = \frac{b}{c}\frac{\pi}{2} = \frac{3k^2}{k^3}\frac{\pi}{2} = \frac{3\pi}{2k} = \frac{3\pi\sigma}{2} = 4.712\ \sigma \qquad \text{(II.14)}$$

The results obtained by our simulation program for this case are clear to the next figure 12 and show the effectiveness of this method. Also we are going to present simulation results and how this method was used for polygon tracking and star tracking. (The reader will be able to find all the necessary codes for finding the leaving point for the various cases in the Appendix of this Thesis).

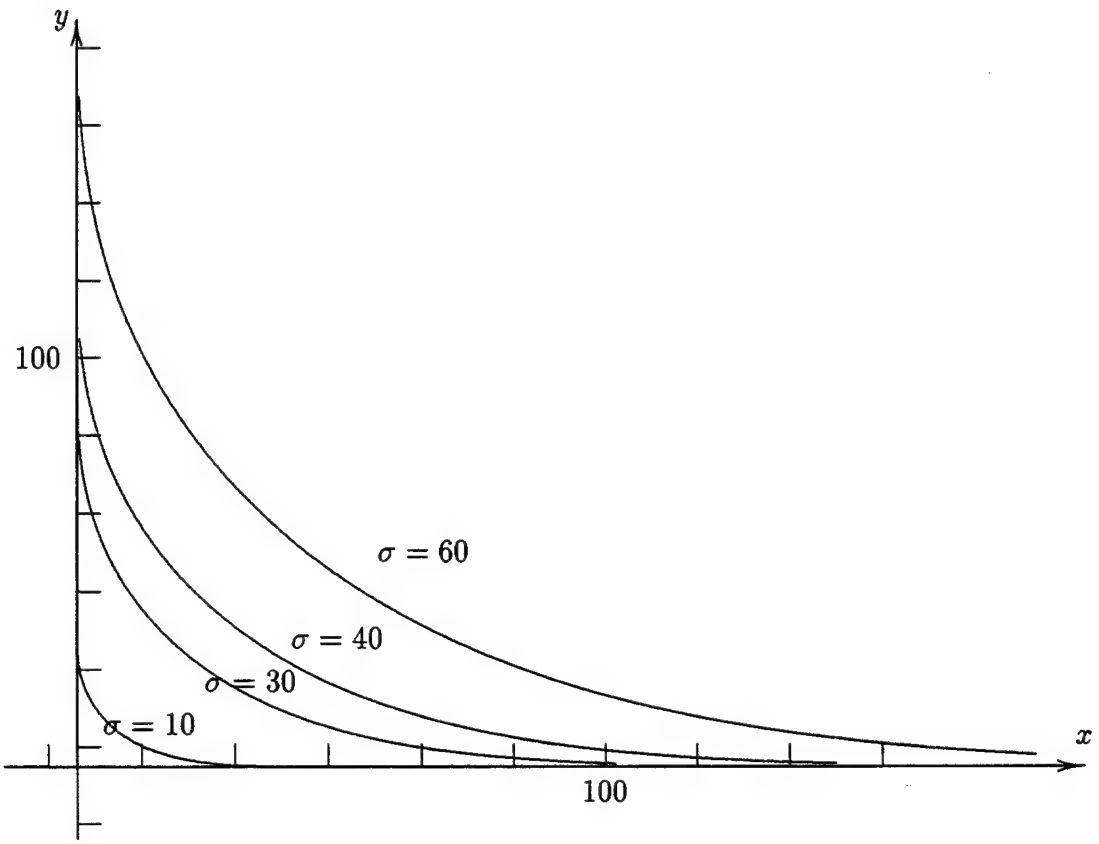



Figure 12. Tracking of X-axis using the *neutral switching method*

Figure 13 shows the simulation results, when the vehicle is tracking the X-axis from different angles:30 degrees, 60 degrees, 90 degrees, 120 degrees and 150 degrees. The initial configuration is $q_0 = ((0.0, 0.0), -\pi/2, 0)$.

As we see, all the above results were obtained for special cases where the vehicle is moving on one of the two axis(X, or Y). Now we are going to examine the more general case where the two lines-path segments- cross each other randomly.

## 1. Intersection of two lines

We consider the general case where we have two lines with configurations

$$q_1 = (x_1, y_1, \theta_1, \kappa_1) \tag{II.15}$$

$$q_2 = (x_2, y_2, \theta_2, \kappa_2) \tag{II.16}$$

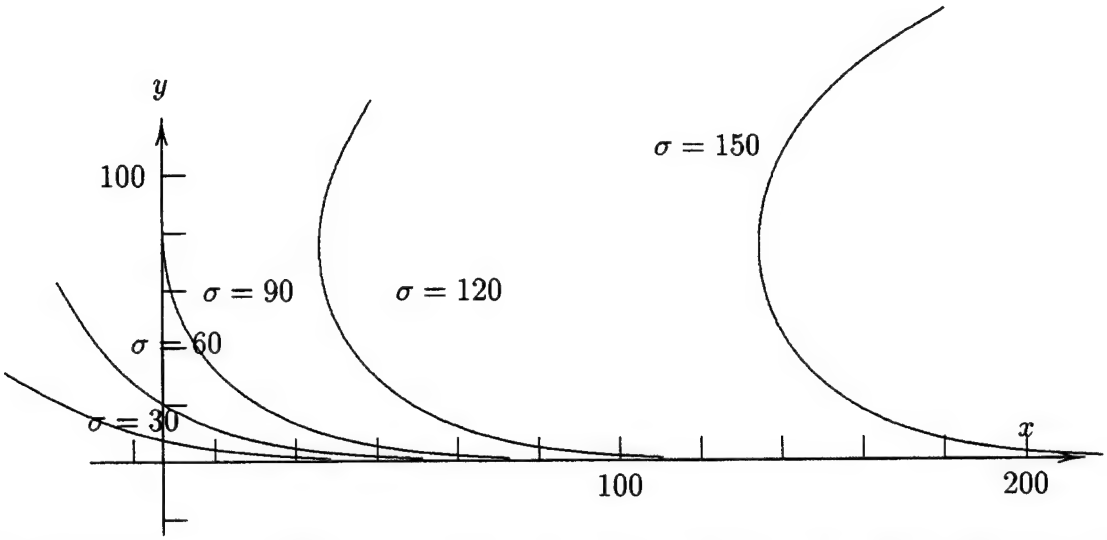



Figure 13. Tracking of X-axis using the *neutral switching method* from different angles

respectively, given $\kappa_1 = \kappa_2 = 0$. In other words the two path segments are straight lines.

Let the intersection of the two lines be $(x, y)$. Then it is obvious that the distance between this point and either of the lines is zero (Figure 14). The calculation of the leaving point is simple and straight forward, as we will see right away.

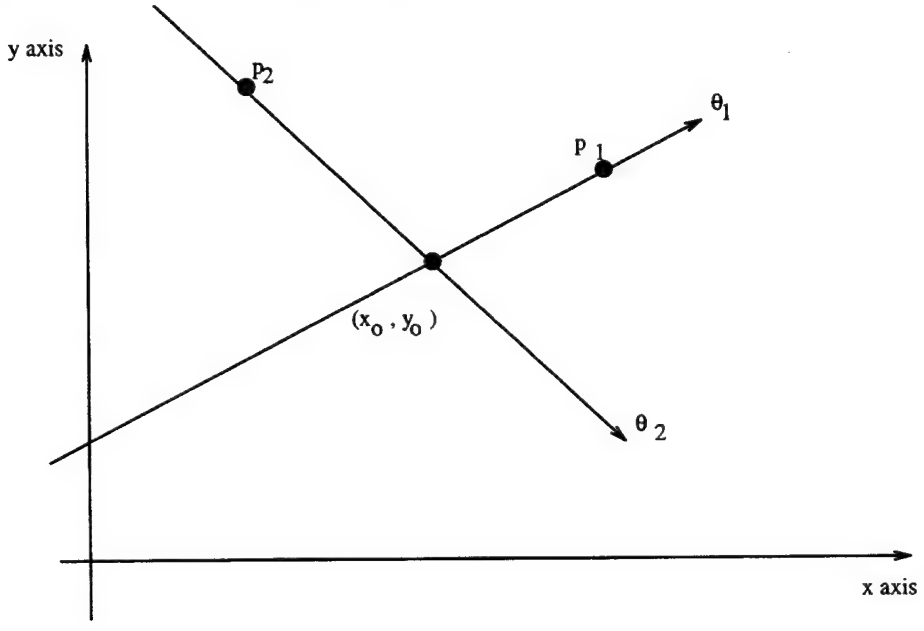



Figure 14. Intersection of two lines

$$(y - y_1)\cos\theta_1 - (x - x_1)\sin\theta_1 \;=\; 0 \qquad \text{(II.17)}$$

$$(y - y_2)\cos\theta_2 - (x - x_2)\sin\theta_2 = 0 \tag{II.18}$$

This is rewritten as

$$\begin{pmatrix} \sin\theta_1 & -\cos\theta_1 \\ \sin\theta_2 & -\cos\theta_2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1\sin\theta_1 - y_1\cos\theta_1 \\ x_2\sin\theta_2 - y_2\cos\theta_2 \end{pmatrix} \tag{II.19}$$

Therefore, if

$$\begin{vmatrix} \sin\theta_1 & -\cos\theta_1 \\ \sin\theta_2 & -\cos\theta_2 \end{vmatrix} = \sin\theta_2\cos\theta_1 - \cos\theta_2\sin\theta_1 = \sin(\theta_2 - \theta_1) \neq 0 \tag{II.20}$$

we can solve the simultaneous equations II.17 and II.18 as follows.

$$\begin{aligned} x &= \frac{1}{\sin(\theta_2 - \theta_1)} \begin{vmatrix} (x_1\sin\theta_1 - y_1\cos\theta_1) & -\cos\theta_1 \\ (x_2\sin\theta_2 - y_2\cos\theta_2) & -\cos\theta_2 \end{vmatrix} \\ &= \frac{-\cos\theta_2(x_1\sin\theta_1 - y_1\cos\theta_1) + \cos\theta_1(x_2\sin\theta_2 - y_2\cos\theta_2)}{\sin(\theta_2 - \theta_1)} \end{aligned} \tag{II.21}$$

$$\begin{aligned} y &= \frac{1}{\sin(\theta_2 - \theta_1)} \begin{vmatrix} \sin\theta_1 & (x_1\sin\theta_1 - y_1\cos\theta_1) \\ \sin\theta_2 & (x_2\sin\theta_2 - y_2\cos\theta_2) \end{vmatrix} \\ &= \frac{\sin\theta_1(x_2\sin\theta_2 - y_2\cos\theta_2) - \sin\theta_2(x_1\sin\theta_1 - y_1\cos\theta_1)}{\sin(\theta_2 - \theta_1)} \end{aligned} \tag{II.22}$$

## 2. Problem Statement

Suppose that we have two lines with configurations

$$q_1 = (x_1, y_1, \theta_1, \kappa_1) \tag{II.23}$$

$$q_2 = (x_2, y_2, \theta_2, \kappa_2) \tag{II.24}$$

respectively.We assume that in our case the curvature values for both lines equal zero.

The main problem then is focused on finding the point $p_{11} = (x_{11}, y_{11})$ on $q_1$ so that $p_{11}$ will become the leaving point for neutral switching. From this point an autonomous mobile vehicle -under the nonholonomic constraint- will use to track the next line. Upon competion of the above calculation we will simulate the motion of

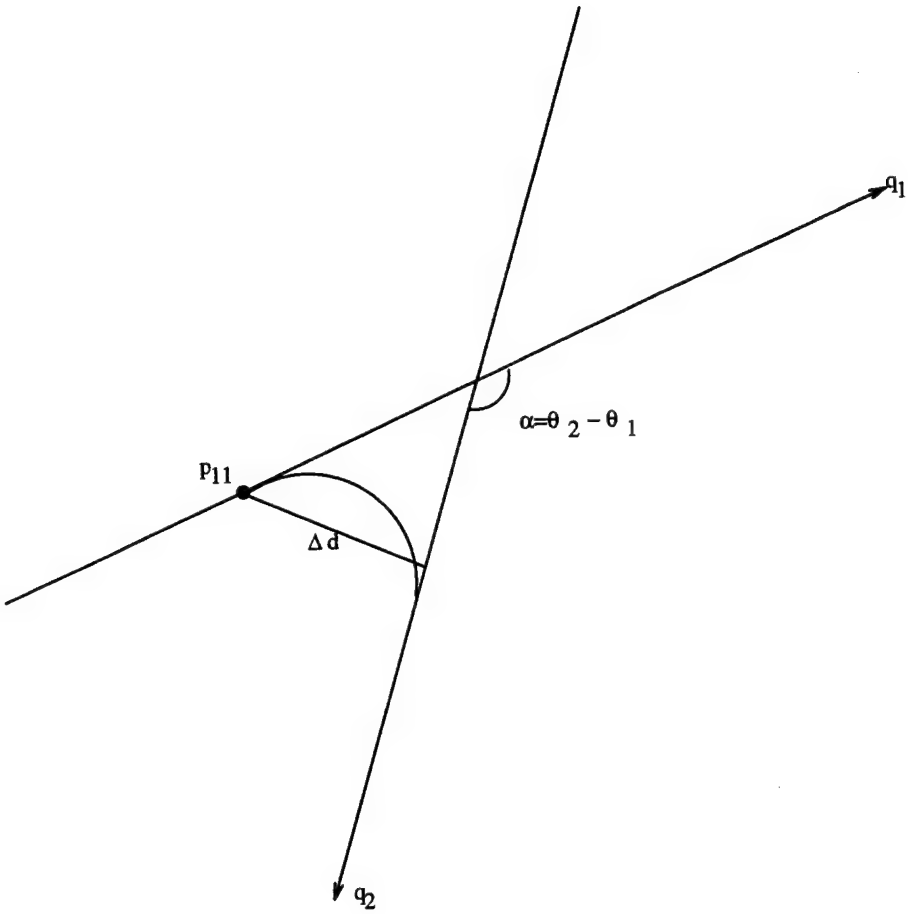



Figure 15. Intersection of two lines in the general case

the robot in C language. The value sigma -$\sigma$-(smoothness) will be given by the user. The leaving point $p_{11} = (x_{11}, y_{11})$ must satisfy two conditions:

**First condition**

$$\frac{d\kappa}{ds} = -(a\Delta\kappa + b\Delta\theta + c\Delta d) = 0$$

In our case

$$k = 0, \text{ and } \theta = \theta_1$$

Also we have

$$\Delta d = \frac{-b(\theta_1 - \theta_2)}{c} = \frac{3k^2(\theta_2 - \theta_1)}{k^3} = 3\sigma(\theta_2 - \theta_1) \qquad \text{(II.25)}$$

On the other hand

$$\Delta d = 0, \quad \text{where } (x^*, y^*) \tag{II.26}$$

is in $q_2'$ 's coordinate system.

$$\Delta d = y^* = (x - x_2)\sin\theta_2 + (y - y_2)\cos\theta_2 \tag{II.27}$$

**Second condition**

The point (x,y) must be on the first directed line. On the other hand $y^* = 0$ on the $q_1$ 's coordinate system.

$$y^* = (x - x_1)\sin\theta_1 + (y - y_1)\cos\theta_1 \tag{II.28}$$

From (1),(2) $\Rightarrow$

$$-x\sin\theta_2 + y\cos\theta_2 = -x_2\sin\theta_2 + y_2\cos\theta_2 + 3\sigma(\theta_2 - \theta_1) \tag{II.29}$$

$$-x\sin\theta_1 + y\cos\theta_1 = -x_1\sin\theta_1 + y_1\cos\theta_1 \tag{II.30}$$

From (3),(4) $\Rightarrow$

$$\begin{pmatrix} sin\theta_1 & +\cos\theta_1 \\ sin\theta_2 & +\cos\theta_2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -x_1\sin\theta_1 + y_1\cos\theta_1 \\ -x_2\sin\theta_2 + y_2\cos\theta_2 + 3\sigma(\theta_2 - \theta_1) \end{pmatrix} \tag{II.31}$$

The next condition also must satisfy:

$$D = \sin\theta_2\cos\theta_1 - \cos\theta_2\sin\theta_1 = \sin(\theta_2 - \theta_1) \neq 0 \tag{II.32}$$

By solving the equations (II.31) we find the coordinates of the point in question:

$$\begin{aligned} x &= \frac{1}{\sin(\theta_2 - \theta_1)} \begin{vmatrix} (-x_1\sin\theta_1 + y_1\cos\theta_1) & +\cos\theta_1 \\ (-x_2\sin\theta_2 + y_2\cos\theta_2 + 3\sigma(\theta_2 - \theta_1) & +\cos\theta_2 \end{vmatrix} \\ &= \frac{\cos\theta_2(-x_1\sin\theta_1 + y_1\cos\theta_1) - \cos\theta_1(-x_2\sin\theta_2 + y_2\cos\theta_2 + 3\sigma(\theta_2 - \theta_1)}{\sin(\theta_2 - \theta_1)} \end{aligned} \tag{II.33}$$

$$\begin{aligned} y &= \frac{1}{\sin(\theta_2 - \theta_1)} \begin{vmatrix} sin\theta_1 & (-x_1\sin\theta_1 + y_1\cos\theta_1) \\ sin\theta_2 & (-x_2\sin\theta_2 + y_2\cos\theta_2 + 3\sigma(\theta_2 - \theta_1) \end{vmatrix} \\ &= \frac{sin\theta_1(-x_2\sin\theta_2 + y_2\cos\theta_2 3\sigma(\theta_2 - \theta_1)) + \sin\theta_2(-x_1\sin\theta_1 + y_1\cos\theta_1)}{\sin(\theta_2 - \theta_1)} \end{aligned} \tag{II.34}$$

As we see, the equation of the steerinf function, governs the behavior of the robot and manipulates its motion, by forcing changes in its position and orientation. The most interesting feature of the *neutral switching method* is that is based also on symmetry. This means that it works and it predicts perfectly the behavior of the robot in all the cases.

We will now discuss now the various cases we will encounter in the real world.We're also going to show, the way this symmetry affects each one of the interactions of the robot in the real world. In other words the symmetry that exists in nature interchanges the terms within the equation, yet leaves the equation the same.

We must now determine the different cases the motion of our robot falls in.

Case 1. This case is depicted very clearly in Figure 16 and is the case where the vehicle should turn when the condition $\frac{dk}{ds} \geq 0$ is fulfilled.

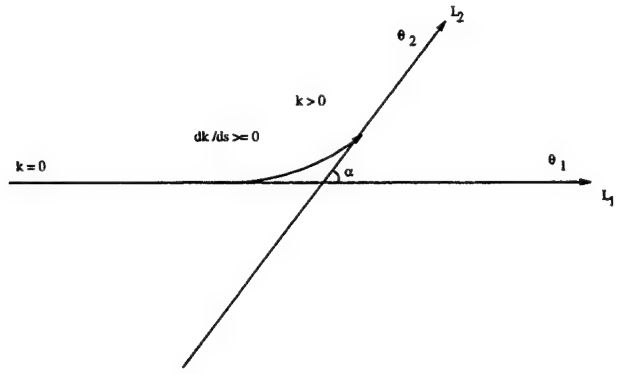



Figure 16. Case where $\kappa > 0$

Case 2. This case is similar to case 1 with the difference that now the curvature $\kappa$ is less than zero. This means that the vehicle has to turn when the condition $\frac{dk}{ds} \leq 0$ is fulfilled.

Case 3. If $\Phi = (\theta_2 - \theta_1) = \pm\pi$ we have the case that is depicted in Figure 18

Case 4.If $\Phi(\theta_2 - \theta_1) = 0$ then either $L_1 = L_2$ or the two lines are parallel as we can see in Figure 19

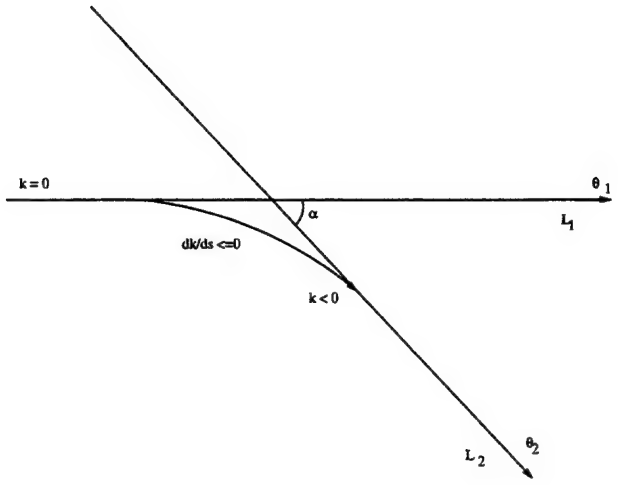



Figure 17. Case where $\kappa < 0$

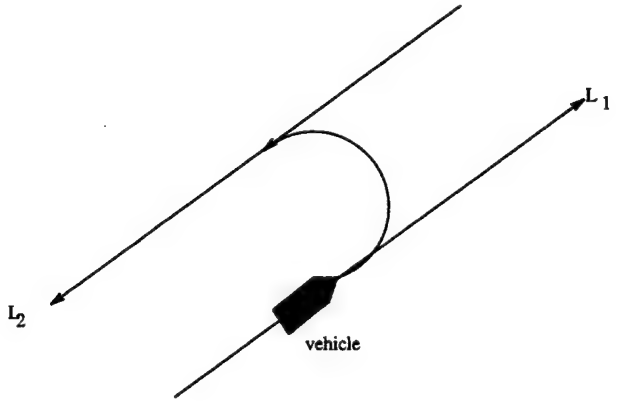



Figure 18. Case where $\Phi = (\theta_2 - \theta_1) = \pm\pi$

## 3. Conclusions

From our discussion so far the reader should be able to distinguish *neutral switching method*'s two essential features:

1. symmetry
2. The ability to explain large amounts of experimental data with the most economical mathematical expressions.

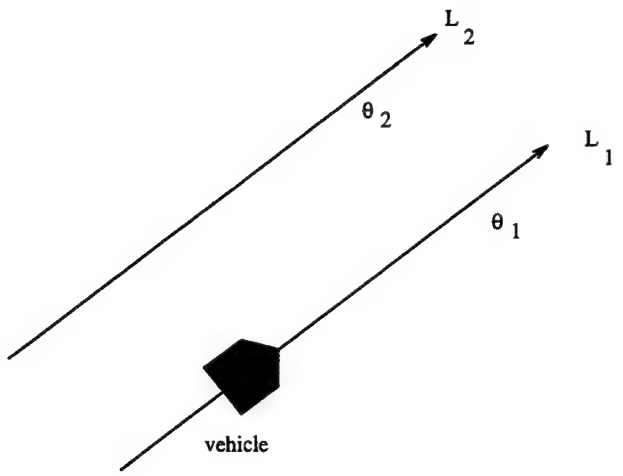



Figure 19. Case where $\Phi = (\theta_2 - \theta_1) = 0$

## D. SIMULATION RESULTS

Here we present more simulation results obtained by the imlementation in C language of neutral switching method.All the codes were tested in UNIX/SUN workstation at the AI laboratory. Those results show clearly the practical aspect and the efficiency of our method.

In Figure 20, the vehicle is supposed to have initial configuration $q_1 = ((0.0, 20.0), 45.0, 0.0)$, and the target line $q_2 = ((30.0, 0.0), -45.0, 0.0)$. The value of smoothness, $\sigma = 10$, while for Figure 21, the value of smoothness, $\sigma = 40$.

The example in Figure 22, shows the result of the trajectory of circular tracking. The center of the circle is at $(0.0, 0.0)$, the radius equals 10, and the value of smoothness, $\sigma = 10$. Our vehicle's configuration is $q_V = ((0.0, -12.5), 0.0, 0.0)$.

In Figure 23, and Figure 24 we can see simulation results for "star" tracking, for smoothness, $sigma = 5$, and smoothness, $sigma = 10$ respectively.

The neutral switching method algorithm described in this thesis has been impemented in MML, and tested on the experimental robot Yamabico-11 in AI lab. We tried various values for the smoothness $\sigma$ and the speed$v$ of the robot. The robot started with initial configuration $q_0 = ((0.0, 0.0), 0.0, 0.0)$ and was supposed to track the line-$q_1$. The results obtained so far show that the algorithm is working well and

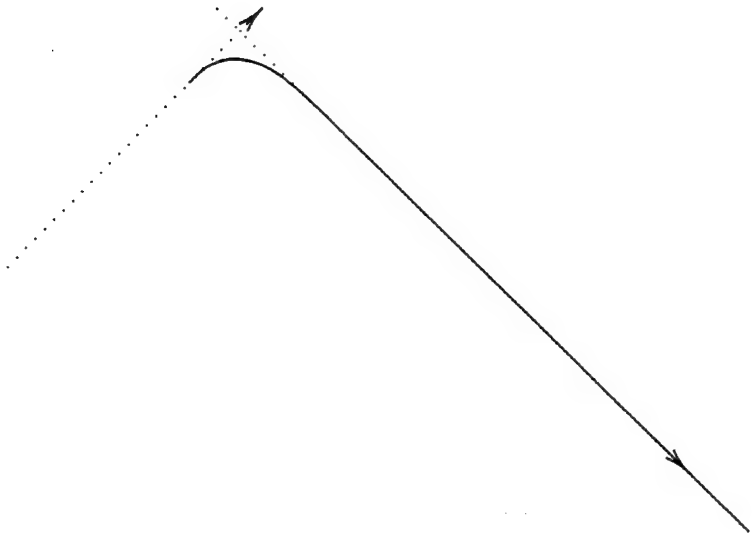


Figure 20. Tracking a target line using *Neutral switching method*

it is very efficient for robot motion planning and motion control.

| $\sigma$ | speed $v$ | $4.712\sigma$ |
|---|---|---|
| 10 | 30 | $q_1 = ((47.12, 0.0), \pi/2, 0.0)$. |
| 20 | 20 | $q_1 = ((94.24, 0.0), \pi/2, 0.0)$. |
| 30 | 10 | $q_1 = ((141.36, 0.0), \pi/2, 0.0)$. |

Table I. Values for speed $v$, and smoothness $\sigma$ used in real time implementation

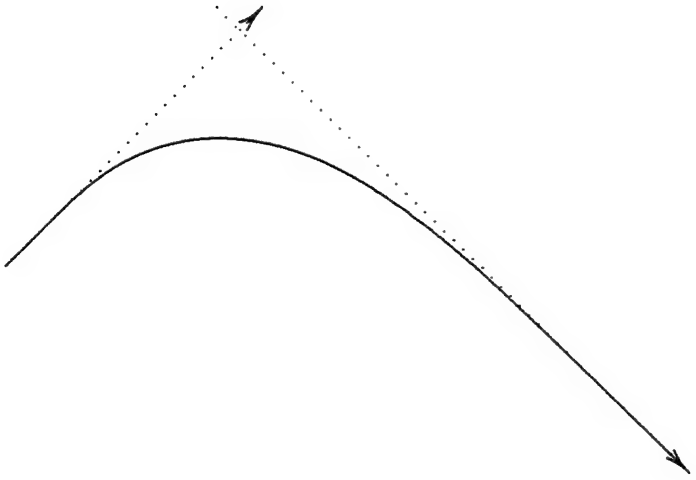


Figure 21. Line tracking using *Neutral switching method*

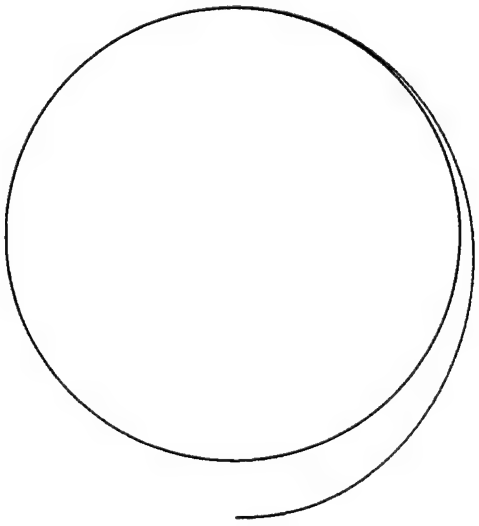


Figure 22. Circle tracking using the *neutral switching method*

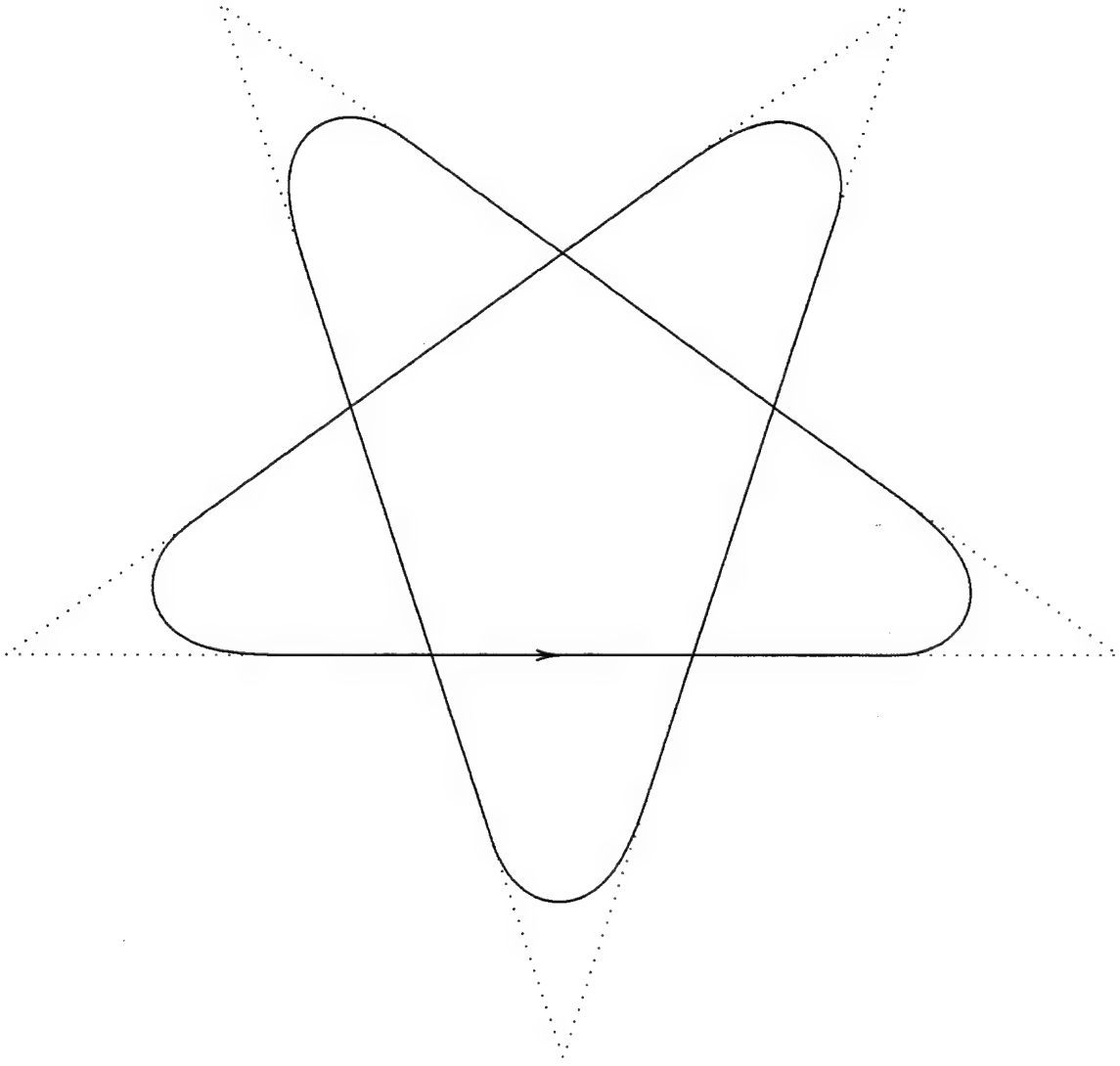


Figure 23. Star motion using the *neutral switching method* (I)

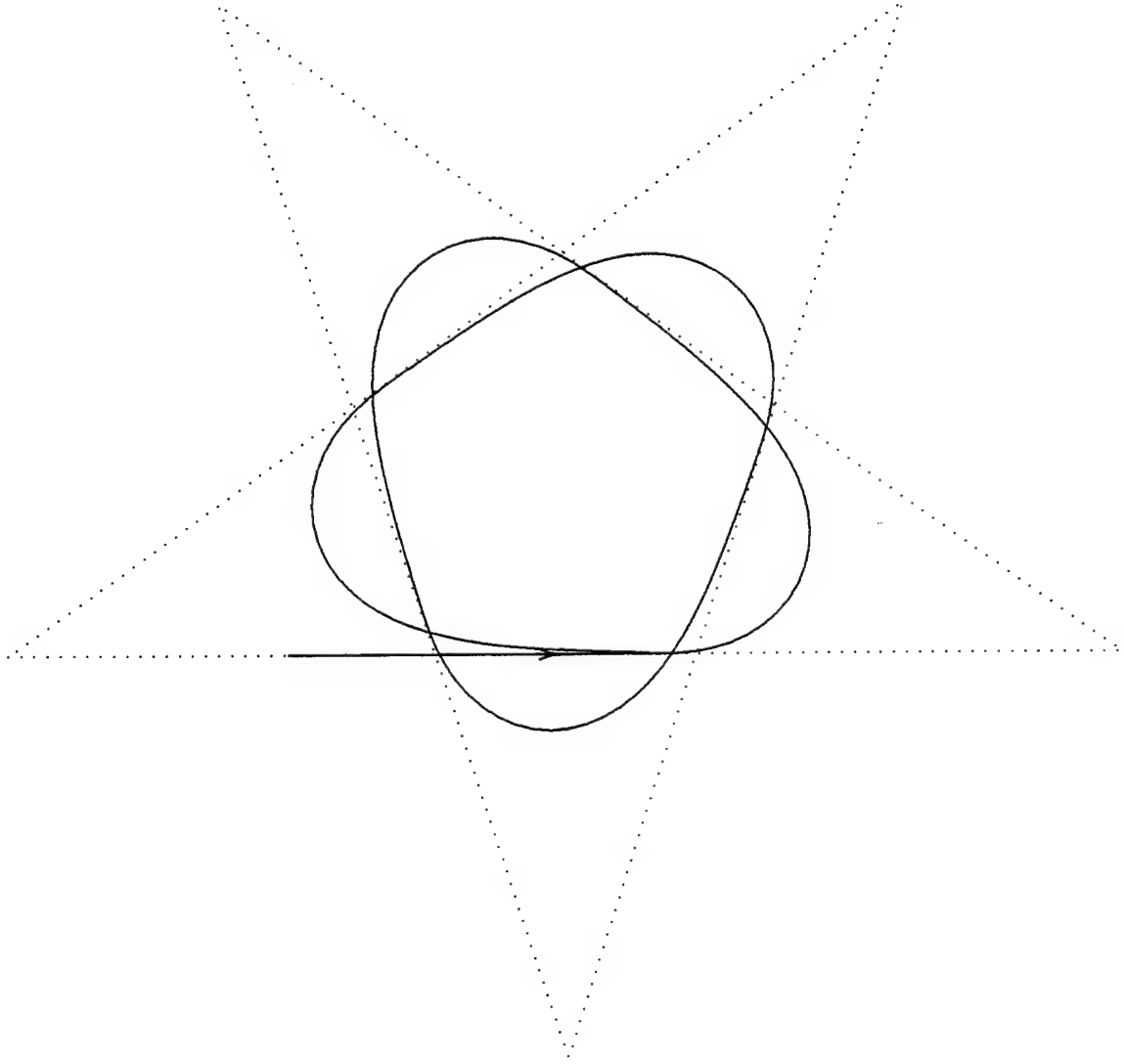


Figure 24. Star motion using the *neutral switching method* (II)

# III. POLYGONS

## A. INTRODUCTION

The purpose of this chapter is to present the various concepts and definitions about polygons, and polygonal worlds. We cover basic concepts which form the basis of our theory as well as the necessary schemas(data structures) we are using to represent a polygon. After that we use the neutral switching method to track a polygon.

### 1. General Definitions

We represent a polygon by an ordered set

$$B = \{v_1, v_2, \cdots, v_n\}, \quad n \geq 3$$

of n distinct points(vertices) in a plane.

As we see from the above definition the simplest polygon is a triangle, since a polygon must have at least three edges. We call a polygon with $n$ vertices a $n$-gon.

The union of all edges in a polygon $B$ is considered to be the value, $val(B)$, of the polygon.

*Definition*: A polygon is called simple if it satisfies the following two conditions(Kanayama)

1. if no triple of vertices $(v, \varphi(v), \varphi^2(v))$ in $B$ are colinear
2. if there is no pair of nonconsecutive edges sharing a point in it

A simple polygon partitions the plane into two disjoint regions, the *bounded*, and *unbounded* ones, separated by the polygon(*Jordan curve theorem*).

In our theory we make the assumption that the first vertex in a polygon $B$ is considered that with the minimum $x$-coordinate among all the vertices.

The direction of an edge $\overline{v, \varphi(v)}$ in a polygon $B$ is defined as

$$\Psi(p_1, p_2) \equiv atan2(y_2 - y_1,\ x_2 - x_1)$$

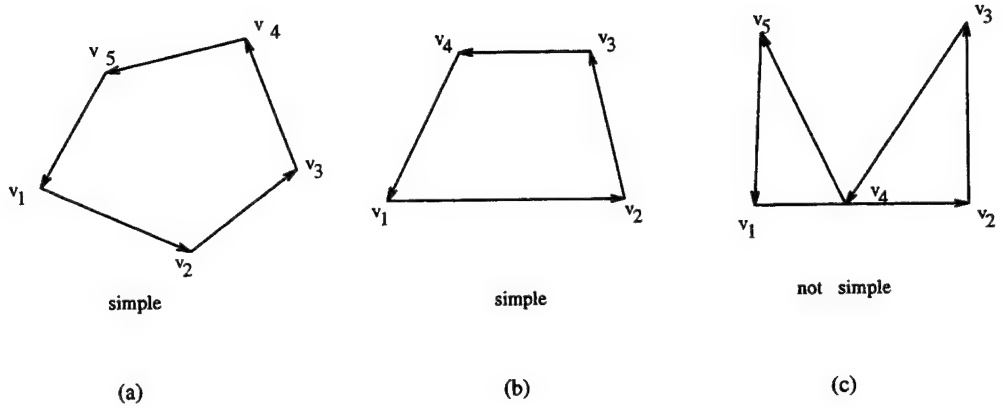



Figure 25. Examples of simple and non-simple polygons (I)

where $p_1 \equiv (x_1, y_1)$ and $p_2 \equiv (x_2, y_2)$ are two distinct points.

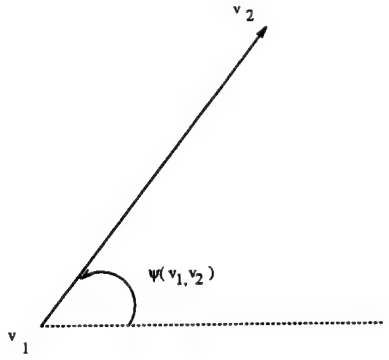



Figure 26. Orientation of an edge

In a polygon $B$ we define an *exterior angle*, $\delta_i$, as the normalized angle between the directions of an edge and its previous one related to vertex $v_i$.

$$\delta_i = \Phi\left(\Psi(v_i, \varphi(v_i)) - \Psi(\varphi^{-1}(v_i), v_i)\right)$$

An exterior angle is positive or negative. It is not equal to 0 or $\pm\ \pi$, because any three consecutive vertices are not colinear. A vertex $v_i$, on a polygon is said to be *convex* if all the vertices in $B$ are convex. Otherwise, it is said to be *concave*.

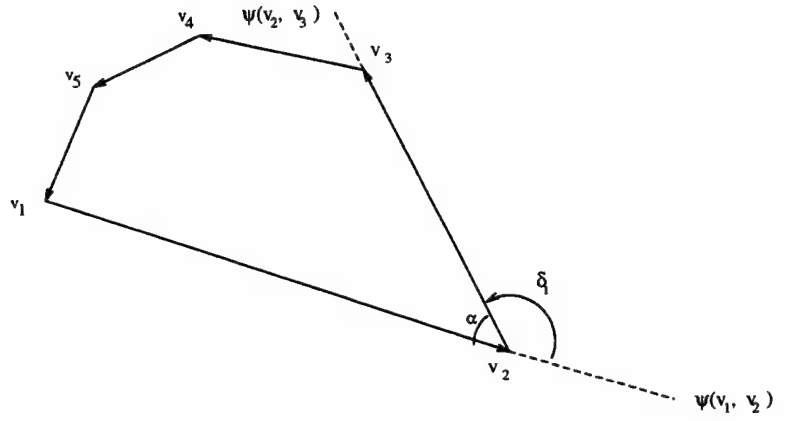



Figure 27. Interior and exterior angle of a simple polygon

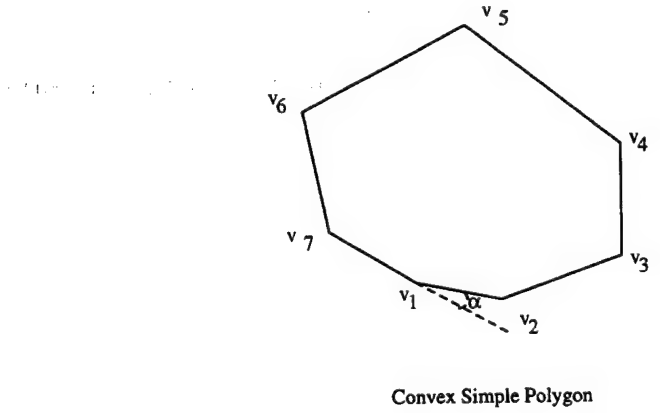



Figure 28. Convex polygon

## 2. Problem Definition

Given $v$, one vertex on a polygon, find a straight path parallel to $\overline{v\varphi(v)}$, which has a clearance of $w_o$ from $\overline{v\varphi(v)}$(by $(a,b,\alpha)$ representation configuration). Track this path by using *neutral switching method.*

The approach we are using for polygon tracking proceeds through the following steps-phases: In order for our method to be clear enough to the reader we reformulated the previous phases in more detailed terms and we decomposed most of the steps into their connected components. This way the solution is developed in subsequent clearer sections.

Phase 1-: Build a polygon $(B)$ world.

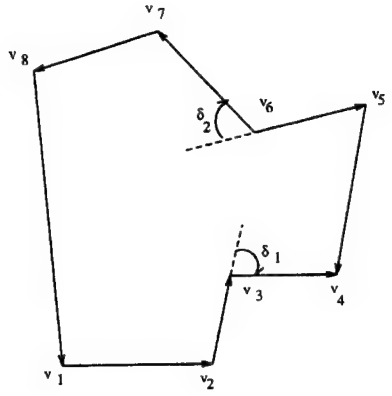



Figure 29. Concave polygon

The world, illustrated in Figure 33, is represented as a linked list of polygons, where each polygon is a doubly linked list of its vertices. Access to the world is gained through the manipulation of a pointer to one of the polygons of the list. (The reader should be able to find all the necessary code for a generarion of a polygonal world in the Appendix).

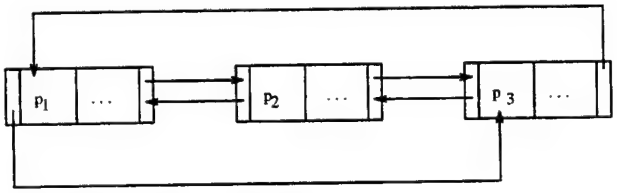



Figure 30. Representation of a world data structure

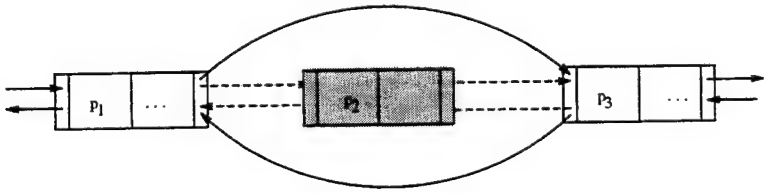



Figure 31. Pointer manipulation for deletion of a node

Phase 2 : Compute one line(ccw or cw) with clearance of $w_o$, given a vertex $v$ in a polygon $B$, of a polygonal world W.

Since our motion planning problem emphasizes ostacle avoidance we compute a safe path for our vehicle. In our case the safest path is one that is on a certain

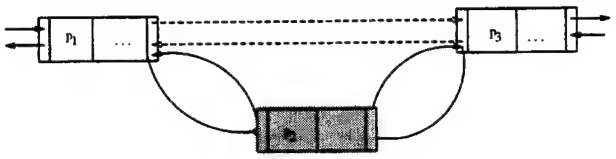


Figure 32. Pointer manipulation for insertion of a node

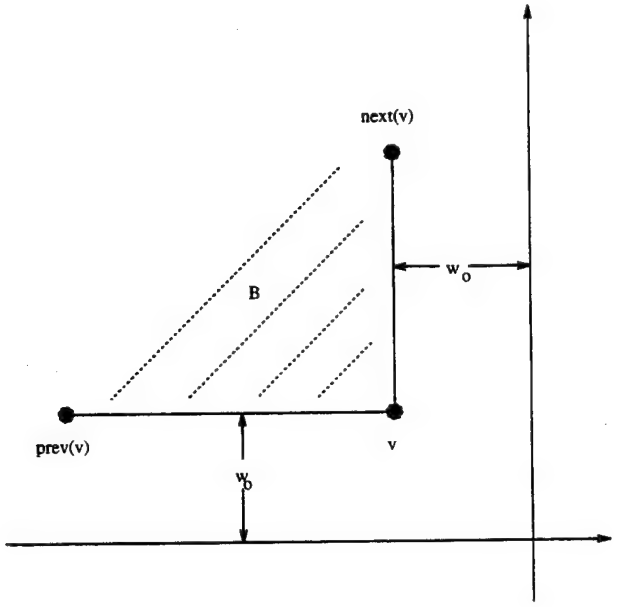



Figure 33. A safety path around a polygon

distance away from the polygon we are going to track. This way the robot departs from the original path, and tracks the computed line by simply specifying the safety distance.

Phase 3 : Compute 5 lines around $B$. The various steps we went through in order to compute the edges of a polygon in our world can be described as follows:

**Tracking an edge** $\overline{v_i v_{i+1}}$
**Input:** $w_o$ desired safety distance

**Output:** line the configuration of a line

**begin**
1. Define the edge configuration $e_i$

$e_i = (v_i, \Psi(v_i, v_{i+1}), 0)$

2. Define the safety distace configuration
   $safe = (0, -w_o, 0)$((−) for ccw, (+) for cw)
3. $line = e_i \circ safe$
4. **return** line

**end**

Phase 4 : Track the lines by using neutral switching method. For the vehicle to exhibit the previously described problem of tracking a computed edge we simply need to specify the desired safety distance. Now its time to show how to specify the desired path sequence.

We followed two methods as we were proceeding to the solution of our problem: (a)**using** $x^*$ as our calculated condition, and (b)**using steer() function**

**1st method**

**begin**
1. **while** {
2. $(x_o, y_o) \longleftarrow$ compute leaving point
3. $(x^*) \longleftarrow$compute $x^*$ switch line $(i = i + 1)$
4. apply *next* function to advance the vehicle one step
5. }

**end**

Figure 34. Pseudocode for Tracking a line using the $x^*$ distance

**2nd method**

In the next section of this chapter we present simulation results of the previous methods. To track a polygon we used the second previously described algorithm(using steer() function) for its simplicity and efficiency.

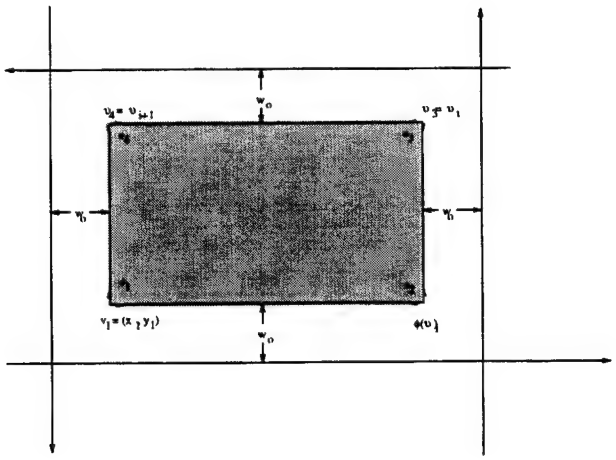


Figure 35. Tracking the edges of a polygon

**begin**
1. **while** i $\leq$ N
2. **if** *steer*() >= 0
3. switch line (i =i + 1)

**end**

Figure 36. Pseudocode for Tracking a line using steer() function

## B. SUMMARY

This chapter develops a method a representation of a polugonal world suitable for the motion of an autonomous robot lika Yamabico. This world as we saw is a polygon with one exterior boundary polygon, and zero or more non-intersecting polygons inside it. After that we showed how to track a specific polygon of this world in a safety distance $w_o$ from it, using the neutral switching method.

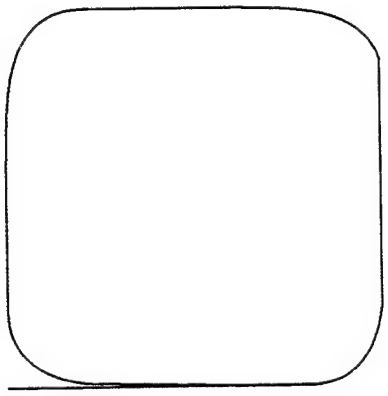


Figure 37. Tracking of four given lines

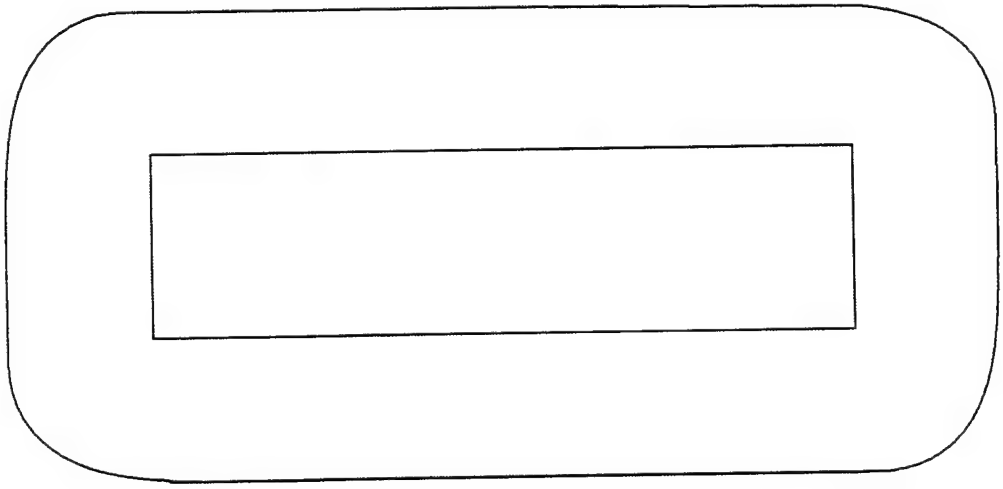


Figure 38. Polygon tracking by using the *neutral switching method*

# IV. HARDWARE AND SOFTWARE ARCHITECTURE OF YAMABICO-11

In this chapter we describe the hardware and software system of the robot—Yamabico-11 which we're using to test our theory experimentally.

## A. HARDWARE SYSTEM

Yamabico-11 (see Figure 40) is an automomous, experimental, wheeled robot which has been developed at the Naval Postgraduate School (NPS) over the last years under the supervision of Professor Yataka Kanayama.

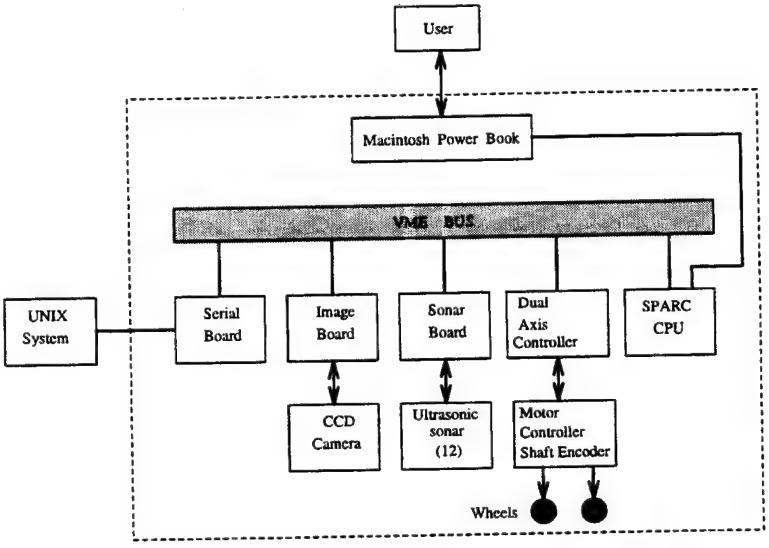



Figure 39. Diagram of Yamabico-11 hardware architecture

### 1. CPU

Yamabico is an autonomous robot and it contains a Sparc CPU which is able to run executable files which have been previously downloaded via ethernet. The CPU is controlled through the Sparc Debug Monitor via an RS-232 connection to an Apple Powerbook. The Ironics IV-SPARC-33 is a single processor, VMEbus Interface, CPU board. It contains a 25 MHz SPARC Integer Unit, a Floating Point Unit, and

a Cache Controller and Memory Management Unit. The card installed in Yamabico has 64 Kbytes of cache, and 16 Mbytes of 80ns DRAM. It provides two RS-232 serial I/O ports, two programmable timers, and seven user-definable LEDs.

The Ironics SPARC board contains 16 Mbytes of physical memory, yet provides 32 bit addresses (4 GBytes). This 4 GBytes address space is logically divided into several regions. The three most important regions are the Local DRAM, Region 3, and Local I/O.

(1)Local DRAM The local DRAM is the physical memory present on the board, and is addressed from 0x00000000 to 0x01000000. The lower limit is fixed, while the upper limit is determined by the amount of physical memory present. This space is used for the kernel and user programs, and for allocating dynamic memory.

(2)Region 3 Region 3 starts at the end of region 2, and extends to the bottom of the EPROM space. The default configuration provides addresses from 0xfc000000 to 0xff0000, however, only the upper boundary is fixed. The lower boundary may be changed by writing to the appropriate register as defined in the Ironics manual. MML11 currently does not change the default address map, but does provide for Region 3 to be VME bus A16 addressable. All devices on the VME bus are addressed from Region 3. Addresses are obtained by adding the 16-bit base address of a specific hardware device to the region 3 offset of 0xfc000000. This includes the shaft encoders, quad serial boards, and sonar board.

(3)Local I/O The local I/O region contains the addresses for the registers which control the operation of the Ironics SPARC board. They are addressed from 0xfff0000 to 0xffffffff.Both limits are fixed. Internal interrupts are those generated on the CPU board. The two most important are the Timer 1 and Timer 2 interrupts. Timer 1 can be set to provide interrupts at 50, 100, or 1000 hz. Currently, MML11 uses Timer 1 to provide the 10ms (100 Hz) motion control interrupt. Timer 2 provides a broader range of interrupts, and is currently unused. The interrupt vectors for Timer 1 and Timer 2 are defined by the Local Interrupt Vector Base Register(0xfffc0057). Yamabico's

communication interface can be accomplished by the help of an input/output subsystem that provides three important functions;screen input/output via the on board laptop computer, facilities for downloading executable programs to Yamabico's main memory, and functions for retrieval of sonar data collected by Yamabico.

External interrupts are those generated off the CPU board. The most important are from the quad serial boards, and the sonar board, which are handled through the 7 VMEbus Interrupt Request lines.

## 2. Wheels

In Yamabico-11, there are six wheels in total. Two wheels, lying on the robot's center line, drive the robot. The remaining four casters are smaller than the drive wheels and each has a shock absorbing material connected to the robot's chassis, two in the front and two in the rear. Two DC motors are used to drive the wheels, one for each drive wheel. There is also a reduction gear box connected between the motor and the drive wheel. The motor control board controls the motors. In Yamabico the control program is called MML system. In this system there is an integer variable called 'pwm.' The pwm value is ranged from $-127$ to 127, its absolute value represents the amount of time the motor will be activated and its sign represents the motor rotational direction. The positive sign is clockwise for right wheel control value(rpwm) and is counterclockwise for left wheel control value(lpwm). The negative sign is just opposite. Each motor is activated by the amount of time which the pwm value represents. If we want the motor to get half of the motor's maximum rotational speed, we will set the pwm value to 64 or -64 and if we want the motor to get all the rotational speed we 'll have to set it to 127 or -127. The real relationship between the pwm value and the actual robot velocity are obtained by experiments. As we see the robot's control program gives us the capability to achieve the velocity we desire and this capability is the very basis of our robot Yamabico-11. The Yamabico-11 hardware architecture is illustrated in Figure 39.

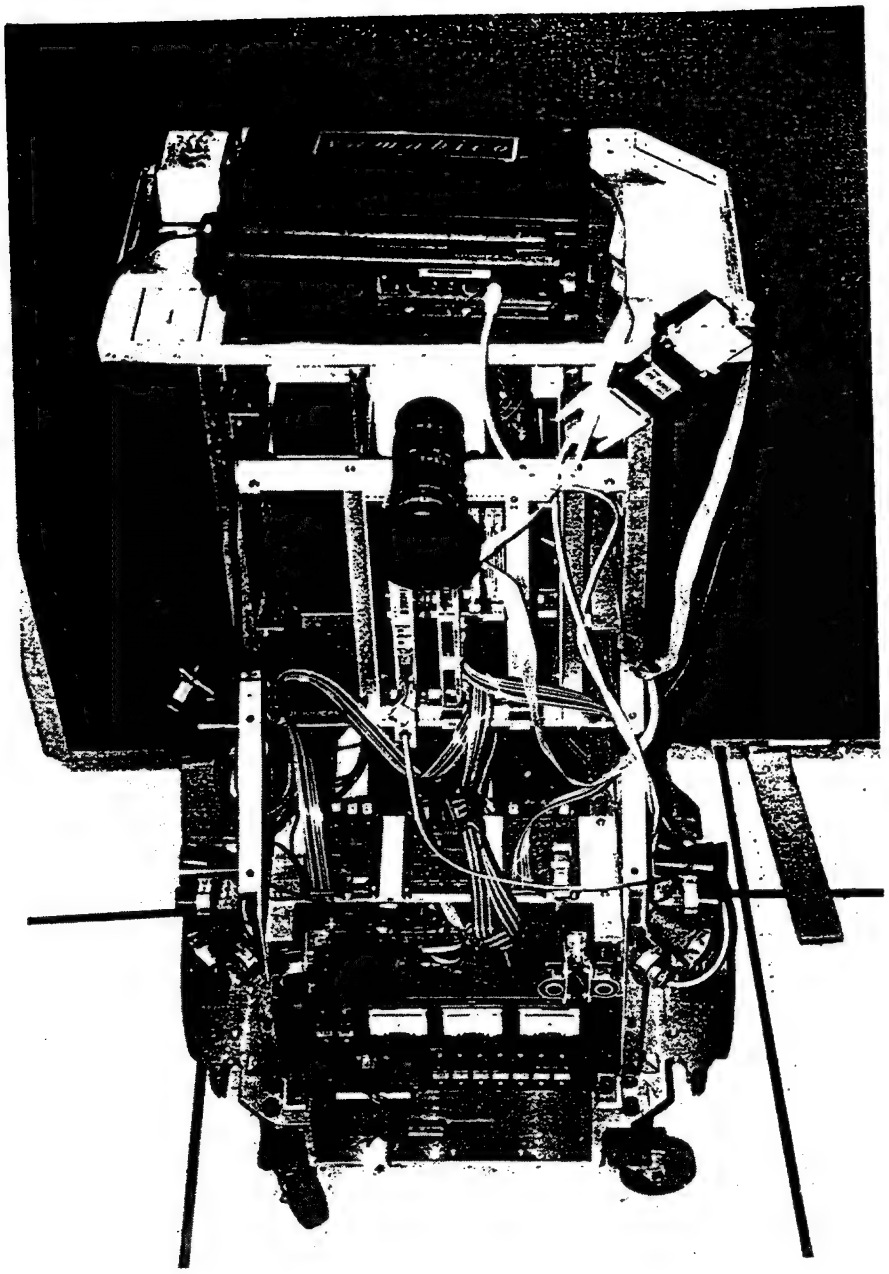


Figure 40. Autonomous mobile robot, Yamabico-11

### 3. Sonars

Sensors include twelve sonar transdusers, precision dead-reckoning wheel encoders, and a color video camera. The Yamabico's software is written in C, and lately a lot of efforts have been made to upgrade the system so it'll support C++.The motion of the robot is achieved by an electric motor powered by batteries.

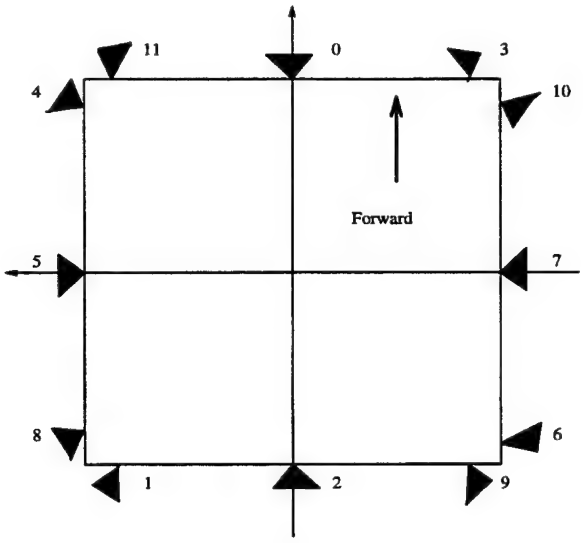



Figure 41. Yamabico-11 sonar configuration

## B. SOFTWARE ARCHITECTURE OF YAMABICO-11

All programs on the robot are developed using a Sun 3/60 workstation and the Unix operating system. These programs are written in the 'C' programming language, compiled and then downloaded to the robot via a RS-232 link at a rate of 19600 baud. The software system consists of an operating system kernel and a user program loaded at different addresses in the robot's main memory. The kernel needs to be downloaded only once during the course of a given experiment. A user program user(), can be modified and downloaded quickly to support rapid development. An on board notebook computer is provided to accomplish command level communication to and from the user. The Model based Mobile-robot Language **MML** is the driving force behind the robot. It is actually a reprogrammable software system and

a multitasking operating system that provides robot motion and sensor functions and also allocates processing resources while performing odometry functions. Tasks are assigned according to their priority level. This is accomplished with the help of the eight interrupt levels that the motorola CPU has.

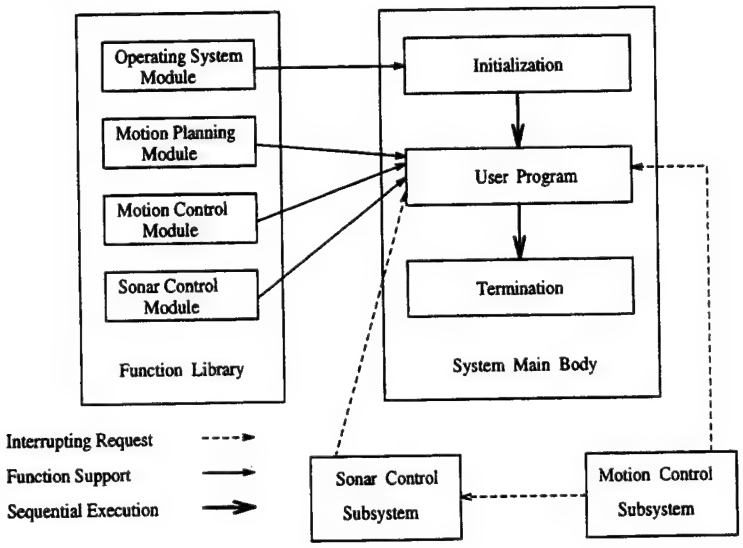



Figure 42. MML-11 software conceptual architecture

## 1. User Program Utility

We distinguish two kind of programs in Yamabico's software structure:system programs and user programs. User programs control the robot's motion in our experimental environment by including all the necessary instructions. An example of a user() program is given in the APPENDIX.

## 2. Library Functions

A special importance should be given to Yamabico's geometric module which provides the necessary mathematical support for the various required spatial reasoning tasks. There are three important components in this susystem;assignment functions for specifying geometric variables,math utility functions for manipulating the geometric variables and path tracking geometric support functions for reasoning about path elements.

## 3. Functions

The path utility functions provide a library of routines for the algebraic manipulation of geometric variables. For example,the composition function is used to perform 2D transformations and the inverse function determines the algebraic inverse of a given configuration. These functions support algebraic manipulations for automatic dead reckoning error correction. Also provided are an assortment of utility functions for spatial reasoning math on board Yamabico. Examples include three types of normalize functions and a ceiling function. All of these utilities support path tracking vehicle control. The path tracking geometric support functions serve to connect individual path elements for smooth vehicle motion. This subsystem is composed of two types of functions which are related.

## 4. Odometry Capability

We should say a few words about how Yamabico's software system maintains its odometry estimate. A set of functions are used to provide automatic vehicle odometry correction. What they're doing actually is setting the initial configuration of the robot, reading the current value of the robot's odometry estimate and updating this value according to the new estimate.

# V. CONCLUSIONS-FUTURE RESEARCH

## A. SUMMARY

We conclude our thesis by summarizing the technique of *neutral switching method* and its contributions to the field of Robotics. We also give in this chapter a final reflection on the long-term aim of our research in Intelligent Robotics.

The *neutral switching method* which was thoroughly presented in Chapter IV was described by the equation (II.5) and is a major contribution toward more autonomous and efficient robots. We could even say more *smart* robots. This method is extremely useful as we saw for the motion control of the autonomous vehicles and enhances their capabilities by the following way:

First, it gives a description of the desired motion of the vehicle in a more transparent way, than any other existing methods.

With our method the motion of the robot is simply described in a more general way. This gives the AI researcher the capability to program and control the behavior of the vehicle more efficiently. The movement of the vehicle along a polygonal world is smoother and all the target tracking maneuveurs are being safer at the desired speed.

Secondly, another contribution of N.S.M is the safe path tracking where a path in our programs is represented by a configuration.

By using the algorithm described in chapter VI we were able to generate a sequence of path segments that represented the desired path of the robot and, we were able to program the robot on a simulator to maneuver on this path and gradually track a polygon.

We also saw how the smoothness $\sigma$ played an important role in our research by determining it as the distance the robot moves along the reference path before it converges to this path. After this we examined what kind of data structures were suitable for representation of a polygonal world. This way we gave the robot the

capability to correctly perceive its surrounding environment and program its behavior.

Autonomous robots like Yamabico, should be enabled by their navigational system to autonomously navigated through various worlds to their required destinations. This autonomous navigation capability can be used in various applications such as unmanned explorations or operations of dangerous environments, manipulation of hazardous materials e.t.c

The problem of robot navigation has been studied by several researchers so far as the reader can very easily see in the list of references used for this thesis.

Robotics as a research area of computer science has been proven to be full of a variety of issues ranging from abstarct mathematical to highly realistic problems.

The algorithm for polygon tracking proposed by Professor Kanayama based on N.S.M has provided many practical advantages for the guiding of the movement of an autonomous vehicle. Fist of all it is reliable and easy to understand and implement. Secondly, it is very well documented and it is supported by a strong mathematical theory. Its contribution to the Robotics science is considered a major one.

## B. FUTURE RESEARCH

As we saw in chapter VI the key notion behind Neutral Switcing Method is selecting an appropriate leaving point from the current path segment our vehicle is moving on to the next one. The method we 're using gives us a unique point for each path combination. The selection of that point is done when the steering function returns a "zero feedback" value:

$$\frac{d\kappa}{ds} = -(a\Delta\kappa + b\Delta\theta + c\Delta d) = 0$$

Another more general and efficient approach would be to consider the leaving point calculated when the folowing condition is considered:

$$\frac{d\kappa}{ds} = -(a\Delta\kappa + b\Delta\theta + c\Delta d) = min$$

Also a future researcher should consider the case where the navigation of autonomous mobile robots like Yamabico, is not always done in worlds whose models are considered known *a priori.*

We expect this Thesis to be useful as a future reference by other students who are going to have the luck to work in Robotics field.

# APPENDIX.

```
#if 0
                        Thesis Research
                        ---------------
Filename          : Line_Tracking
Author            : Karamanlis Vasilios
Operating System  : Unix
Description       : This file contains all the necessary functions to compute
                    the leaving point for neutral switching used in line tracking.

#endif


#include <stdio.h>
#include <math.h>

#define S0      0.0
#define PI      3.1415926536
#define SQR(x)  x*x
#define QUB(x)  x*x*x
#define S_Max   500

FILE       *f1;

typedef struct
   {
    double x;
    double y;
   }POINT;

typedef struct
   {
   POINT p ;
   double theta;
   double kappa;
   }CONFIGURATION;


/*************************************************************************
```

```
Function name : Normalize()
Purpose       : This function is used to get the value of sigma from the user
Parameters    : -
Return Type   : double
*****************************************************************************/

double Normalize(double angle)
{
   while ( angle > PI )
     {
       angle -= 2*PI;
     }
   while ( angle < -PI)
   {
       angle += 2*PI;
     }
   return angle;
}


/*****************************************************************************
Function name : Get_Circ_Transformation
Purpose       : This function is used to compute the circular transformation
Parameters    : &delta_theta , &delta_s
Return Type   : CONFIGURATION
*****************************************************************************/

CONFIGURATION Get_Circular_Transformation(double delta_s,double delta_theta)
{
   CONFIGURATION q;

   q.p.x = (1.0 - ((delta_theta * delta_theta)/6.0) +
   ((delta_theta*delta_theta*delta_theta*delta_theta)/120.0)) * delta_s;
   q.p.y = (0.5 - ((delta_theta * delta_theta)/24.0) +
   ((delta_theta*delta_theta*delta_theta*delta_theta)/720.0)) * delta_theta
   * delta_s;
   q.theta = delta_theta;

   return (q);
}
```

```
/*********************************************************************
Function name : next()
Purpose       : apply steering function , get next configuration
Parameters    : &delta_theta, &delta_s, double sigma,double s
Return Type   : CONFIGURATION
*********************************************************************/

CONFIGURATION next(double s,double sigma ,double delta_s ,double delta_theta ,
CONFIGURATION q1,CONFIGURATION q2)
{
   double k,a,b,c,lamda;

   k = 1.0/sigma;
   a = 3 * k;
   b = 3 * SQR(k);
   c = QUB(k);

   lamda = - (a * q1.kappa + b *Normalize(q1.theta - q2.theta) +
         c *(-(q1.p.x-q2.p.x)*sin(q2.theta)+(q1.p.y-q2.p.y)*cos(q2.theta)));
   q1.kappa = q1.kappa + lamda * delta_s;
   delta_theta = q1.kappa * delta_s;
   q2 = Get_Circular_Transformation(delta_s,delta_theta );

   q1.p.x = q1.p.x + q2.p.x * cos(q1.theta) - q2.p.y * sin(q1.theta);
   q1.p.y = q1.p.y + q2.p.x * sin(q1.theta) + q2.p.y * cos(q1.theta);
   q1.theta = q1.theta + q2.theta;

   return (q1);
}


/*********************************************************************
Function name : Get_Input_For_q1()
Purpose       : This function prompts the user to enter the initial
                    configuration for q1.
Parameters    : -
Return Type   : CONFIGURATION
*********************************************************************/

CONFIGURATION Get_Input_For_q1()
{
  CONFIGURATION q1 ;
```

```
  double x1,y1,theta1,k1;

  printf("\nEnter the value for x1 please :");scanf("%lf",&x1);
  printf("\nEnter the value for y1 please:");scanf("%lf",&y1);
  printf("\nEnter the value of the angle theta (deg)for q1 please :");
  scanf("%lf",&theta1);
  printf("\nEnter the value of the curvature for q1 please :");
  scanf("%lf",&k1);

  q1.p.x = x1;
  q1.p.y = y1;
  q1.theta = theta1*PI/180;
  q1.kappa = k1;

  return q1;
}


/*********************************************************************
Function name : Get_Input_For_q2()
Purpose       : This function prompts the user to enter the initial
                  configuration for q2.
Parameters    : -
Return Type   : CONFIGURATION
*********************************************************************/

CONFIGURATION Get_Input_For_q2()
{
  CONFIGURATION q2;
  double x2,y2,theta2,k2;

  printf("\nNow enter the value for x2 please :");scanf("%lf",&x2);
  printf("\nNow enter the value for y2 please :");scanf("%lf",&y2);
  printf("\nNow enter the value of the angle for q2 please :");
  scanf("%lf",&theta2);
  printf("\nEnter The value of the curvature for q2 please :");
  scanf("%lf",&k2);

  q2.p.x = x2;
  q2.p.y = y2;
  q2.theta = theta2*PI/180;
  q2.kappa = k2;
```

```
  return q2;
}


/*********************************************************************
Function name : Leave
Purpose       : This function is used to calculate the leaving point
Parameters    : CONFIGURATION,CONFIGURATION
Return Type   : double x,double y
*********************************************************************/

POINT Leave(CONFIGURATION q1,CONFIGURATION q2,double sigma)
{
   POINT temp;
   double N,T;

   N=Normalize(q2.theta-q1.theta);
   T=sin(N);

   temp.x=(cos(q2.theta)*((-q1.p.x*sin(q1.theta)+(q1.p.y*cos(q1.theta))))-
          cos(q1.theta)*((-q2.p.x*sin(q2.theta)+(q2.p.y*cos(q2.theta)+
          (3*sigma*N)))))/T;
   temp.y=(-sin(q1.theta)*((-q2.p.x*sin(q2.theta)+(q2.p.y*cos(q2.theta)+
          (3*sigma*N))))+
          sin(q2.theta)*((-q1.p.x*sin(q1.theta)+(q1.p.y*cos(q1.theta)))))/T;

   return temp;
}


/*********************************************************************
Function name : main()
Purpose       : This function is the main program
Parameters    : -
Return Type   : int
*********************************************************************/

int main()
{
   double s;
   double delta_s,delta_theta ,sigma;
```

```
   CONFIGURATION q1,q2;

   f1 = fopen("s20150.dat","w");

   q1 = Get_Input_For_q1();
   q2 = Get_Input_For_q2();

   printf("\nEnter the value of sigma please :");scanf("%lf",&sigma);
   printf("\nEnter the value of delta_s please :");scanf("%lf",&delta_s);

   q1.p= Leave(q1,q2,sigma);
   printf("\n x=%f y=%f\n",q1.p.x,q1.p.y);

   for(s=0;s<= S_Max;s+= delta_s)
      {
          fprintf(f1,"\n%15.10f  %15.10f",q1.p.x , q1.p.y);
          q1 = next(s,sigma,delta_s,delta_theta,q1,q2);
      }

   fclose(f1);

   printf("\nOk!Now let's see the results.\n");
   printf("\nType 'gnuplot' please\n");
}

#if 0
                        Thesis Research
                        ---------------
Filename          : Tracking_of_Lines
Author            : Karamanlis Vasilios
Operating System  : Unix
Description       : This file contains all the necessary functions for tracking
                    four lines forming a rectangular(intermediate step for polygon
                    tracking)using the new method 'neutal switching'

#endif


#include <stdio.h>
#include <math.h>

#define PI      3.1415926536
```

```
#define SQR(x)  x*x
#define QUB(x)  x*x*x

FILE       *f1;

typedef struct
   {
    double x;
    double y;
   }POINT;

typedef struct
   {
   POINT p ;
   double theta;
   double kappa;
   }CONFIGURATION;


/*************************************************************************
Function name : Normalize()
Purpose       : This function is used to get the value of sigma from the user
Parameters    : -
Return Type   : double
*************************************************************************/
double Normalize(double angle)
{
   while ( angle > PI )
     {
       angle -= 2*PI;
     }
   while ( angle < -PI)
     {
       angle += 2*PI;
     }

   return angle;
}


/*************************************************************************
Function name : Get_Circ_Transformation
```

```
Purpose        : This function is used to compute the circular transformation
Parameters     : &delta_theta , &delta_s
Return Type    : CONFIGURATION
*****************************************************************************/

CONFIGURATION Get_Circular_Transformation(double delta_s,double delta_theta)
{
   CONFIGURATION q;

   q.p.x = (1.0 - ((delta_theta * delta_theta)/6.0) +
   ((delta_theta*delta_theta*delta_theta*delta_theta)/120.0)) * delta_s;
   q.p.y = (0.5 - ((delta_theta * delta_theta)/24.0) +
   ((delta_theta*delta_theta*delta_theta*delta_theta)/720.0)) *
   delta_theta * delta_s;
   q.theta = delta_theta;

   return (q);
}


/*****************************************************************
Function name : next()
Purpose        : apply steering function , get next configuration
Parameters     : double s,double sigma ,double delta_s ,
                 double delta_theta ,CONFIGURATION q1,CONFIGURATION q2)
Return Type    : CONFIGURATION
*****************************************************************/

CONFIGURATION next(double s,double sigma ,double delta_s ,
             double delta_theta ,CONFIGURATION q1,CONFIGURATION q2)
{
  CONFIGURATION q3;
  double k,a,b,c,lamda;

  k = 1.0/sigma;
  a = 3 * k;
  b = 3 * SQR(k);
  c = QUB(k);

  lamda = - (a * q1.kappa + b *Normalize(q1.theta - q2.theta) +
     c *(-(q1.p.x-q2.p.x)*sin(q2.theta)+(q1.p.y-q2.p.y)*cos(q2.theta)));
```

```
  q1.kappa = q1.kappa + lamda * delta_s;
  delta_theta = q1.kappa * delta_s;
  q2 = Get_Circular_Transformation(delta_s,delta_theta);

  q1.p.x = q1.p.x + q2.p.x * cos(q1.theta) - q2.p.y * sin(q1.theta);
  q1.p.y = q1.p.y + q2.p.x * sin(q1.theta) + q2.p.y * cos(q1.theta);
  q1.theta = q1.theta + q2.theta;

  return (q1);
}


/*********************************************************************
Function name      : Get_Input_For_Lines()
Purpose            : This function prompts the user to enter the initial
                         configuration for q1.
Parameters         : -
Return Type        : CONFIGURATION
*********************************************************************/

CONFIGURATION Get_Input_For_Lines()
{
  CONFIGURATION q1 ;
  double x1,y1,theta1,k1;

  printf("\nEnter the value for x please :");scanf("%lf",&x1);
  printf("\nEnter the value for y please:");scanf("%lf",&y1);
  printf("\nEnter the value of the angle theta (deg)for the line please :");
  scanf("%lf",&theta1);
  printf("\nEnter the value of the curvature for the line please :");
  scanf("%lf",&k1);

  q1.p.x = x1;
  q1.p.y = y1;
  q1.theta = theta1*PI/180;
  q1.kappa = k1;

  return q1;
}


/*********************************************************************
```

```
Function name     : Leave
Purpose           : This function is used to calculate the
                        leaving point
Parameters        : CONFIGURATION,CONFIGURATION
Return Type       : double x,double y
*****************************************************************/

POINT Leave(CONFIGURATION q1,CONFIGURATION q2,double sigma)
{
   POINT temp;
   double N,T;

   N=Normalize(q2.theta-q1.theta);
   T=sin(N);

   temp.x=(cos(q2.theta)*((-q1.p.x*sin(q1.theta)+(q1.p.y*cos(q1.theta))))-
          cos(q1.theta)*((-q2.p.x*sin(q2.theta)+(q2.p.y*cos(q2.theta)+
          (3*sigma*N)))))/T;
   temp.y=(-sin(q1.theta)*((-q2.p.x*sin(q2.theta)+(q2.p.y*cos(q2.theta)+
          (3*sigma*N))))+
         `sin(q2.theta)*((-q1.p.x*sin(q1.theta)+(q1.p.y*cos(q1.theta)))))/T;

   return temp;
}


/*****************************************************************
Function      : main()
Purpose       : This function is the main program
Parameters    : -
Return Type   : int
*****************************************************************/

int main()
{
   double s, xx;
   double delta_s,delta_theta,sigma;
   int i = 0;
   const int N=4;
   CONFIGURATION qo,qv,q1;
   CONFIGURATION q[N+2];
```

```
f1 = fopen("p.dat","w");

for(i=0;i<N;i++)
{
   q[i]=Get_Input_For_Lines();
}

/* necessary for the tracking to close nicely by corresponding
   q[0]=q[4]and q[1]=q[5] */
q[N]=q[0];
q[N+1]=q[1];
/* qv is the actual point we're on */
qv=q[0];

/* printing the starting point */
printf("\nThe coordinates of the starting point are :\n");
printf("\nqv: x=%f y=%f \n",qv.p.x,qv.p.y);

printf("\nEnter the value of sigma please :");scanf("%lf",&sigma);
printf("\nEnter the value of delta_s please :");scanf("%lf",&delta_s);

/* calculating the first leaving point */
qo.p= Leave(q[0],q[1],sigma);
qo.theta = q[0].theta;
qo.kappa = q[0].kappa;

/* printing the coordinates of the first leaving point qo are :\n");
printf("\n x=%f y=%f ",qo.p.x,qo.p.y);

i = 0;
while(i<=N+1)
{
   fprintf(f1,"\n%15.10f  %15.10f",qv.p.x , qv.p.y);
   xx = (qo.p.x-qv.p.x)*cos(qv.theta)+(qo.p.y-qv.p.y)*sin(qv.theta);
      if(xx < 0.0)
      {
         i++; /* switching lines */
         qo = q[i];
         qo.p= Leave(q[i],q[i+1],sigma);
         qo.theta = q[i].theta;
         qo.kappa = q[i].kappa;
```

```
            if (i<4)
     printf("\nThe coordinates for the line i=%d x=%f y=%f\n",
                    i+1,qo.p.x,qo.p.y);
          }
       qv = next(s,sigma,delta_s,delta_theta,qv,qo);
    }

    fclose(f1);

    printf("\nOk!Now let's see the results.\n");
    printf("\nType 'gnuplot' please\n");
}

#if 0

                                  Thesis Research
                                 -----------------
Filename          : Build_of_Polygon.c
Author            : Karamanlis Vasilios
Operating System  : Unix
Description       : This file contains all the necassary functions to build a
                    polygon, using old data structures(not Lombardo's)

#endif


#include <stdio.h>
#include <math.h>
#include <stdlib.h>

FILE    *f4;

typedef struct {
   double x;
   double y;
   }
POINT;

typedef struct {
   POINT point;
   double theta;
   double kappa;
```

```
   }
CONFIGURATION;

typedef struct Vertex{
      POINT          coord;
      int            name;
      int            convex;
      struct Vertex  *Next;
      struct Vertex  *Prev;
      }
Vertex;

typedef struct Polygon{
      int                name;
      int                mode;
      struct Vertex      *VertexList;
      struct Polygon     *Next;
      }
Polygon;

typedef struct {
      int             name;
      Polygon         *PolygonList;
      }
World;

double  WorldData[] =
{1.0, 0.0, 4.0,
      153.2, 118.0,
      397.2, 118.0,
      397.2, 182.0,
      153.2, 182.0
};


/*******************************************************
Function name : OpenFile
Purpose       : This function is used to open a file
Parameters    : -
Return Type   : -
*******************************************************/
```

```
void OpenFile()
{
   f4= fopen("p.dat", "w");
}


/*****************************************************************
Function name : buildWorld
Purpose       : This function is used to build a polygon
Parameters    : -
Return Type   : World
*****************************************************************/

World*
buildWorld(World *curWorld)
{

      int        Vertexcount;
      int        polycount;
      double     tempX, tempY ,tempxo ,tempyo;

      Polygon    *curPoly;
      Vertex     *curVertex;
      Vertex     *PrevVertex;
      Vertex     *NextVertex;
      Vertex     *FirstVertex;
      int        i, numvert, numpoly;
      int        nvert, nconvexvert;

      Vertexcount = 0;
      polycount = 0;
      nconvexvert = 0;
      i = 0;

      curWorld = (World*)malloc(sizeof(World));
      curPoly = curWorld->PolygonList =
  (Polygon *) malloc(sizeof(Polygon));
      numpoly = (int) WorldData[i++];

      /* loop until all Polygons are read */
      while (1) {
    curPoly->name = (int) WorldData[i++];
```

```
    numvert = nvert = (int) WorldData[i++];
            polycount++;
    curVertex = curPoly->VertexList =
(Vertex *) malloc(sizeof(Vertex));

            tempxo=WorldData[3];
            tempyo=WorldData[4];

           /* loop until all vertices for this
      Polygon are read */
            while (numvert > 0) {
  tempX = WorldData[i++];
  tempY = WorldData[i++];
                  Vertexcount++;
  curVertex->name = Vertexcount;
  curVertex->coord.x = tempX;
  curVertex->coord.y = tempY;
  if(curPoly->name ==0)
                   fprintf(f4,"\n %f %f",curVertex->coord.x,
                            curVertex->coord.y);
  numvert--;

                  /* last Vertex read for this Polygon */
                  if(numvert == 0)
    {
                      curVertex->Next = curPoly->VertexList;
curPoly->VertexList->Prev = curVertex;
PrevVertex = curVertex;
curVertex = FirstVertex = curPoly->VertexList;
NextVertex = curVertex->Next;
                       fprintf(f4,"\n %f %f",tempxo,tempyo);
                    }
             else
        {
                  curVertex->Next = (Vertex *) malloc(sizeof(Vertex));
                  PrevVertex = curVertex;
                  curVertex = curVertex->Next;
                  curVertex->Prev = PrevVertex;
             }
            }

    numpoly--;
```

```
    if (numpoly <= 0) {
       curPoly->Next = NULL;
       return curWorld;
    }
              else {
          curPoly->Next = (Polygon *) malloc(sizeof(Polygon));
          curPoly = curPoly->Next;
          Vertexcount = 0;
          nconvexvert = 0;
       }

    }

            return curWorld;
       }
```

```
/*****************************************************************
Function name : main
Purpose       : This function is the main file
Parameters    : -
Return Type   : -
*****************************************************************/

int main(void)
{
  World  *curWorld;
  OpenFile();
  curWorld=buildWorld(curWorld);
  fclose(f4);
}


#if 0

                                Thesis Research
                               -----------------

Filename         : Polygon.c
Author           : Karamanlis Vasilios
Operating System : Unix
Description      : This file contains all the necessary functions to build
                   a polygon (using Lombardo's data structures)and create
```

```
                path segments at a given safety distance from it.After
                that it uses the neutral switching method to track it.
                (It uses :"Leave" function and x* distance)

#endif

#include <stdio.h>
#include <math.h>

/* manifest constants */
#define FAILURE      -1
#define PI 3.1415926536
#define SQR(x)  x*x
#define QUB(x)  x*x*x
FILE *f1;

/* typedefs */
typedef struct point {
  double x,y;
} Point;

typedef struct vertex {
  Point point;
  struct vertex
    *previous,
    *next;
} Vertex;

typedef struct {
   Point point;
   double theta;
   double kappa;
   }CONFIGURATION ;

typedef struct polygon {
  int degree;
  Vertex *vertex_list;          /* this is the first vertex */
  struct polygon
    *previous,
    *next;
} Polygon;
```

```
typedef struct world {
  int degree;
  Polygon *poly_list;
} World;


/*********************************************************
Function name : OpenFile
Purpose       : This function is used to open a file
Parameters    : -
Return Type   : -
*********************************************************/

void OpenFile()

{
   f1= fopen("p1.dat","w");
}


/*************************************************************************
Function name:  fatal()
Parameters    :  char *message
Purpose       :  on fatal error print message then print system message
                 then exit
Called by     :  create_world() <world.c>
         create_polygon()         <world.c>
         add_vertex_to_polygon() <world.c>
Calls         :  NONE
*************************************************************************/

void fatal(message)
     char *message;
{
  fprintf(stderr, "Fatal error occured: ");
  perror(message);
  exit(FAILURE);
}


/*************************************************************************
Function    :   create_world()
```

```
Purpose     :   create instance of a world
Returns     :   World *
Called by   :   ANYBODY
Calls       :   fatal() <utilities.c>
Comments    :   this function allocates space for a world and returns a pointer
*****************************************************************************/

World *create_world()
{
  World *w;

  /* allocate memory for a world */
  if((w = (World *)malloc(sizeof(World))) == NULL) {

    fatal("create_world: malloc\n");
    exit(FAILURE);
  }

  /* initialize fields */
  w->degree = 0;
  w->poly_list = NULL;

  return(w);
}


/*****************************************************************************
Function name    : Composition
Purpose          : This function is used to compute the composition
                   of two given transformations
Parameters       : two two dimensional coordinate transformations
Return Type      : CONFIGURATION-the composition of the two
                   transformations q1 and q2
*****************************************************************************/

CONFIGURATION composition(q1,q2)
CONFIGURATION q1;
CONFIGURATION q2;
{
   CONFIGURATION q3;
   q3.point.x = q1.point.x + q2.point.x * cos(q1.theta) - q2.point.y
              * sin(q1.theta);
```

```
   q3.point.y = q1.point.y + q2.point.x * sin(q1.theta) + q2.point.y
                * cos(q1.theta);
   q3.theta = q1.theta + q2.theta;

   return q3;
}


/*************************************************************************
Function name : Get_Circ_Transformation
Purpose       : This function is used to compute the circular
                transformation
Parameters    : &delta_theta , &delta_s
Return Type   : CONFIGURATION
*************************************************************************/

CONFIGURATION Get_Circular_Transformation(delta_s,delta_theta)
double delta_s,delta_theta;
{
   CONFIGURATION q;

   q.point.x = (1.0 - ((delta_theta * delta_theta)/6.0) +
               ((delta_theta*delta_theta*delta_theta*delta_theta)/120.0))
               * delta_s;
   q.point.y = (0.5 - ((delta_theta * delta_theta)/24.0) +
               ((delta_theta*delta_theta*delta_theta*delta_theta)/720.0))
               * delta_theta * delta_s;
   q.theta = delta_theta;

   return (q);
}


/*************************************************************************
Function name : Normalize()
Purpose       : This function is used to get the value of sigma from the user
Parameters    : -
Return Type   : double
*************************************************************************/
double Normalize(angle)
double angle;
{
```

```
   while ( angle > PI )
     {
       angle -= 2*PI;
     }
   while ( angle < -PI)
     {
       angle += 2*PI;
     }

   return angle;
}


/*********************************************************************
Function name : next()
Purpose       : apply steering function , get next configuration
Parameters    : double s,double sigma ,double delta_s ,
                double delta_theta ,CONFIGURATION q1,CONFIGURATION q2)
Return Type   : CONFIGURATION
*********************************************************************/

CONFIGURATION next(sigma ,delta_s ,delta_theta ,q1,q2)
double sigma ,delta_s ,delta_theta ;
CONFIGURATION q1,q2;
{
  CONFIGURATION q3;
  double k,a,b,c,lamda;

  k = 1.0/sigma;
  a = 3 * k;
  b = 3 * SQR(k);
  c = QUB(k);

  lamda = - (a * q1.kappa + b *Normalize(q1.theta - q2.theta) +
     c *(-(q1.point.x-q2.point.x)*sin(q2.theta)+
           (q1.point.y-q2.point.y)*cos(q2.theta)));

  q1.kappa = q1.kappa + lamda * delta_s;
  delta_theta = q1.kappa * delta_s;
  q2 = Get_Circular_Transformation(delta_s,delta_theta);

  q1.point.x = q1.point.x + q2.point.x * cos(q1.theta) - q2.point.y
```

```
                * sin(q1.theta);
  q1.point.y = q1.point.y + q2.point.x * sin(q1.theta) + q2.point.y
                * cos(q1.theta);
  q1.theta = q1.theta + q2.theta;

  return (q1);
}


/*********************************************************************
   Function name      : Leave
   Purpose            : This function is used to calculate the
                        leaving point
   Parameters         : CONFIGURATION,CONFIGURATION
   Return Type        : double x,double y
*********************************************************************/

Point Leave(q1,q2,sigma)
CONFIGURATION q1,q2;
double sigma;
{
   Point temp;
   double N,T;

   N=Normalize(q2.theta-q1.theta);
   T=sin(N);

   temp.x=(cos(q2.theta)*((-q1.point.x*sin(q1.theta)+
          (q1.point.y*cos(q1.theta))))-
          cos(q1.theta)*((-q2.point.x*sin(q2.theta)+
          (q2.point.y*cos(q2.theta)+(3*sigma*N)))))/T;
   temp.y=(-sin(q1.theta)*((-q2.point.x*sin(q2.theta)+
          (q2.point.y*cos(q2.theta)+(3*sigma*N))))+
          sin(q2.theta)*((-q1.point.x*sin(q1.theta)+
          (q1.point.y*cos(q1.theta)))))/T;

   return temp;
}


/*********************************************************************
Function   :   create_polygon()
```

```
Purpose     :   create instance of a polygon
Returns     :   Polygon *
Called by   :   ANYBODY
Calls       :   fatal() <utilities.c>
Comments    :   this function allocates space for a polygon and
                returns a pointer to it
*****************************************************************************/

Polygon *create_polygon()
{
  Polygon *p;

  /* allocate memory for a polygon */
  if((p = (Polygon *)malloc(sizeof(Polygon))) == NULL) {

    fatal("create_polygon: malloc\n");
    exit(FAILURE);
  }

  /* initialize fields */
  p->degree = 0;
  p->vertex_list = NULL;
  p->previous = NULL;
  p->next = NULL;

  return(p);
}


/*****************************************************************************
Function      :   add_vertex_to_polygon(x, y, p)
Parameters    :   double x x coordinate of new vertex
                  double y y coordinate of new vertex
          Polygon *p pointer to an existing polygon
Purpose       :   add a vertex to an existing polygon
Returns       :   void
Called by     :   ANYBODY
Calls         :   fatal() <utilities.c>
Comments      :   adds vertex to the end of the vertex list - no way to insert
                  NOTE: polygon must exist before adding vertices
*****************************************************************************/
```

```
void add_vertex_to_polygon(x, y, p)
     double x, y;
     Polygon *p;
{
  int i; /* loop variable */
  Vertex
    *new_vertex, /* pointer to the new vertex */
    *current_vertex; /* pointer to the current vertex */

  /* allocate space for the new vertex */
  if((new_vertex = (Vertex *)malloc(sizeof(Vertex))) == NULL) {
    fatal("add_vertex_to_polygon: malloc\n");
    exit(FAILURE);
  }

  /* install coordinates */
  new_vertex->point.x = x;
  new_vertex->point.y = y;

  /* check if first vertex */
  if(p->degree == 0) {
    new_vertex->previous = new_vertex;
    p->vertex_list = new_vertex;
  }
  else {

    /* set up the links */
    current_vertex = p->vertex_list;
    current_vertex->previous = new_vertex;

    /* find the last vertex */
    for(i = 1; i < p->degree; i++)
      current_vertex = current_vertex->next;

    new_vertex->previous = current_vertex;
    current_vertex->next = new_vertex;

  }

  new_vertex->next = p->vertex_list;
  p->degree++;
}
```

```
/*****************************************************************************
Function  :   display_vertices_of_polygon(p)
Parameters:   Polygon *p pointer to an existing polygon
Purpose   :   display the vertices of a polygon of the existing world
Returns   :   void
Called by :   ANYBODY
Calls     :   NONE
Comments  :   -
*****************************************************************************/

void display_vertices_of_polygon(p)
Polygon *p;
{

   Vertex *current, *first;

   first=current= p->vertex_list;
   do{
     printf("\n%f %f\n",current->point.x , current->point.y);
     current = current->next;
   }while(current != first);
   return;
}


/*****************************************************************************
Function  :   add_polygon_to_world(p, w)
Parameters:   Polygon *p pointer to an existing polygon
              World *w pointer to an existing world
Purpose   :   add an existing polygon to an existing world
Returns   :   void
Called by :   ANYBODY
Calls     :   NONE
Comments  :   adds polygon to end of polygon list - no way to insert polygon
*****************************************************************************/

void add_polygon_to_world(p, w)
     Polygon *p;
     World *w;
{
```

```
  Polygon *current_polygon; /* pointer to current polygon */

  /* check if first polygon */
  if(w->degree == 0) {
    w->poly_list = p;
  }

  else {

    /* find the last polygon */
    current_polygon = w->poly_list;
    while(current_polygon->next != NULL)
      current_polygon = current_polygon->next;

    p->previous = current_polygon;
    current_polygon->next = p;
  }

  w->degree++;
}


/*****************************************************************************
Function  :   Find_Path_Segment(v)
Parameters:   vertex
Purpose   :   computes the path segment according to a given safety distance
Returns   :   CONFIGURATION

Called by :   ANYBODY
Calls     :   NONE
Comments  :   safety distance is -50 by default(- shows ccw polygon,+ cw )
*****************************************************************************/

CONFIGURATION Find_Path_Segment(v)
Vertex *v;
{
CONFIGURATION safe;
CONFIGURATION edge;

safe.point.x=0.0;
safe.point.y=-50.0;
safe.theta=0.0;
```

```
safe.kappa=0.0;
edge.point.x=v->point.x;
edge.point.y=v->point.y;
edge.theta= atan2(v->next->point.y - v ->point.y,
                          v->next->point.x - v->point.x);
edge.kappa=0.0;

return(composition(edge,safe));
}


/*****************************************
Function name :   main()
Purpose       :   this is the main function
*****************************************/

int main()
{
  Vertex *new_vertex;
  Polygon  *p1;
  World *w;
  Vertex *v,*first;
  double X,Y,s,xx;
  double delta_s,delta_theta,sigma;
  int i=0;
  int N=4;
  CONFIGURATION e,q[6],qo,qv;

  /* make a world */
  w = create_world();

  /* make a polygon */
  p1 = create_polygon();

  /* add three vertices */
  add_vertex_to_polygon(153.2, 118.0, p1);
  add_vertex_to_polygon(397.2, 118.0, p1);
  add_vertex_to_polygon(397.2, 182.0, p1);
  add_vertex_to_polygon(153.2, 182.0, p1);

  display_vertices_of_polygon(p1);
```

```
/* attach polygon to world */
add_polygon_to_world(p1, w);

OpenFile();

v=first = p1->vertex_list;
do{
   q[i] = Find_Path_Segment(v);
   X = q[i].point.x ;
   Y = q[i].point.y ;
   printf("%f %f\n",X,Y);
   v = v->next;
    i++;
   }while( v != first);
 q[N]=q[0];
 q[N+1]=q[1];
 /* qv is the actual point we're on */
 qv=q[0];

 printf("\nEnter the value of sigma please :");scanf("%lf",&sigma);
 printf("\nEnter the value of delta_s please :");scanf("%lf",&delta_s);

 /* calculating the first leaving point */
 qo.point= Leave(q[0],q[1],sigma);
 qo.theta = q[0].theta;
 qo.kappa = q[0].kappa;

 i=0;
 while(i<=N+1)
 {
   fprintf(f1,"\n%15.10f  %15.10f",qv.point.x ,qv.point.y);
   /*printf("\n%15.10f  %15.10f",qv.point.x , qv.point.y);*/
    xx = (qo.point.x-qv.point.x)*cos(qv.theta)+(qo.point.y-qv.point.y)
         *sin(qv.theta);
       if(xx < 0.0)
       {
          i++; /* switching lines */
          qo = q[i];
          qo.point= Leave(q[i],q[i+1],sigma);
          qo.theta = q[i].theta;
          qo.kappa = q[i].kappa;
     - }
```

```
      qv = next(sigma,delta_s,delta_theta,qv,qo);

   }
   fclose(f1);
  return 0;
}


#if 0
                              Thesis Research
                             -----------------
Filename            : main4.c
Author              : Karamanlis Vasilios
Project #3          : Circle Tracking
Operating System    : Unix
Description         : This file contains all the necessary functions for
                      circle tracking
                      (Using the Neutral Switching method)
#endif

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define S0 0.0
#define S_MAX 600.0
#define PI 3.1415926536
#define RAD 180.0/PI

FILE *f1 ;

/* Data structures for defining coordinates X , Y and angle theta */

typedef struct
  {
  double X;
  double Y;
  }POINT;

typedef struct
  {
```

```
  POINT point;
  double theta;
  double kappa;
  }CONFIGURATION;


/*************************************************************************
Function name : Get_Circ_Transformation
Purpose       : This function is used to compute the circular transformation
Parameters    : &delta_theta , &delta_s
Return Type   : CONFIGURATION
*************************************************************************/

CONFIGURATION Get_Circular_Transformation(double ds,double dt)
{
   CONFIGURATION q;
   q.point.X = (1.0 - (dt * dt)/6.0) *ds;
   q.point.Y = (0.5 - (dt * dt)/24.0)*dt*ds;
   q.theta = dt;

   return (q);
}


/*************************************************************************
Function name : Print_to_File
Purpose       : This function is used to write the data to the output file
Parameters    : f,q
Return Type   : -
*************************************************************************/

void Print_to_File(FILE *f,CONFIGURATION q)
{
   fprintf(f,"\n%f %f" , q.point.X , q.point.Y);
}


/*************************************************************************
Function name : Normalize()
Purpose       : This function is used to get the value of sigma from the user
Parameters    : -
Return Type   : double
```

```
**************************************************************************/

double Normalize(double angle)
{
   while ( angle > PI )
     {
       angle -= 2*PI;
     }
   while ( angle < -PI)
   {
       angle += 2*PI;
     }
   return angle;
}


/**************************************************************************
Function name : sgn()
Purpose       : This function is used to return the sign of a value
Parameters    : double
Return Type   : -
**************************************************************************/

int sgn(double v)
{
   int SIGN;
   if(v==0.0)SIGN = 0.0;
      else if(v>0.0)SIGN = 1;
         else SIGN = -1;

   return (SIGN);
}


/**************************************************************************
Function name : Circle_Tracking()
Purpose       : This function is used to get the next configuration
Parameters    : double s,double sig,double ds,double dt,CONFIGURATION q1,q2
Return Type   : CONFIGURATION
**************************************************************************/

CONFIGURATION Circle_Tracking(double s,double sigma,double ds,double dt,
```

```
                                CONFIGURATION q1,CONFIGURATION q2)
{
   CONFIGURATION qc;
   double k,k1,a,b,c,A,E,D,D1,rad,N1,N;
   int SGN;
   sigma=0.5;
   rad=10.0;
   k=1.0/sigma;
   k1=1/rad;
   a=3*k;
   b=3*(k*k)-(k1*k1);
   c=(k*k*k)-3*k*(k1*k1);

   SGN = sgn(rad);
   for(s=S0;s<=200;s+=ds)
   {
      Print_to_File(f1,q1);
      E=atan2((q2.point.Y-q1.point.Y),(q2.point.X-q1.point.X));
      D1=(q2.point.X-q1.point.X)*(q2.point.X-q1.point.X)+(q2.point.Y
         -q1.point.Y)*(q2.point.Y-q1.point.Y);
      D=sqrt(D1);
      N1=q1.theta-(E-(SGN*PI/2));
      N=Normalize(N1);
      A=-(a*(q1.kappa-1.0/rad)+b*N+c*(rad-SGN*D));
      q1.kappa=q1.kappa+A*ds;
      dt=q1.kappa*ds;
      qc=Get_Circular_Transformation(ds,dt);

      q1.point.X = q1.point.X +qc.point.X*cos(q1.theta) - qc.point.Y
                   *sin(q1.theta);
      q1.point.Y = q1.point.Y +qc.point.X*sin(q1.theta) + qc.point.Y
                   *cos(q1.theta);
      q1.theta = q1.theta +qc.theta;
    }
return (q1);
}


/*****************************************************************
Function name : main()
Purpose       : main program
Parameters    : -
```

```
Return Type    : int
*****************************************************************/

int main()
{
   /* declarations of the variables */
   double s;
   double delta_s,delta_theta,sigma;
   CONFIGURATION q1,q2,qpoint;

   /* declares the values used in the program */
   delta_s=0.5; /* value for the length */
   q1.point.Y=0.0; /* Y value of the vehicle */
   /*q1.point.X=-14.53;*/ /* X value of the vehicle for sigma=1.0*/
   /*q1.point.X=-16.8;*/ /* X value of the vehicle for sigma=1.5*/
   /*q1.point.X=-19.2;*//* X value of the vehicle for sigma=2.0*/
   q1.point.X=-12.3;/* X value of the vehicle for sigma=0.5*/
   q1.theta=0.0; /* theta value of the vehicle */
   q1.kappa=0.0; /* kappa value of the vehicle */
   qpoint.point.X=.0; /* X value of the center of the circle */
   qpoint.point.Y=0.0; /* Y value of the center of the circle */
   qpoint.theta=0.0; /* theta value of the center of the circle */
   qpoint.kappa=0.0; /* kappa value of the center of the circle */

   /* prints the current date and time */
   time_t now;
   now = time(NULL);
   printf("The current date and time is : %s",ctime(&now));
   /* welcomes the user to the program */
   printf("\nWelcome to my Circle tracking program in C.\n");
   printf("\nYou are going to deal with advanced Robotics material.\n");
   printf("\nI think you are a little bit nervous.\n");
   printf("\nIt's not a big deal,just only ADVANCED ROBOTICS\n");
   printf("\nLet's start!\n");

   /* opens the output file */
   f1=fopen("path.dat","w");

   /* makes the circle tracking */
   q2=Circle_Tracking(s,sigma,delta_s,delta_theta,q1,qpoint);
   Print_to_File(f1,q2);
   /* closes the output file*/
```

```
   fclose(f1);

   /* prompts the user to use 'gnuplot' drawing program*/
   printf("\nOk!Now let's see the results\n");
   printf("\nType 'gnuplot' please.\n");

}

#if 0

                                Thesis Research
                               -----------------

Filename         : Polygon.c
Author           : Karamanlis Vasilios
Operating System : Unix
Description      : This file contains all the necessary functions to
                   build a polygon and create path segments at a given
                   safety distance from it.After that it uses
                   ''Neutral switcing method'' to track it.

#endif

#include <stdio.h>
#include <math.h>


/* manifest constants */
#define FAILURE      -1
#define PI 3.1415926536
#define SQR(x)  x*x
#define QUB(x)  x*x*x

FILE *f1;

/* typedefs */
typedef struct point {
  double x,y;
} Point;

typedef struct vertex {
  Point point;
  struct vertex
```

```
    *previous,
    *next;
} Vertex;


typedef struct {
   Point point;
   double theta;
   double kappa;
   }CONFIGURATION ;

typedef struct polygon {
  int degree;
  Vertex *vertex_list;          /* this is the first vertex */
  struct polygon
    *previous,
    *next;
} Polygon;

typedef struct world {
  int degree;
  Polygon *poly_list;
} World;


/*********************************************************
Function name : OpenFile
Purpose       : This function is used to open a file
Parameters    : -
Return Type   : -
*********************************************************/

void OpenFile()

{
   f1= fopen("poly5.dat","w");
}


/*************************************************************************
Function  : fatal()
Parameters: char *message
```

```
Purpose   : on fatal error print message then print system message then exit
Called by : create_world() <world.c>
    create_polygon() <world.c>
    add_vertex_to_polygon() <world.c>
Calls     : NONE
*****************************************************************************/

void fatal(message)
     char *message;
{
  fprintf(stderr, "Fatal error occured: ");
  perror(message);
  exit(FAILURE);
}


/*****************************************************************************
Function :   create_world()
Purpose  :   create instance of a world
Returns  :   World *
Called by:   ANYBODY
Calls    :   fatal() <utilities.c>
Comments :   this function allocates space for a world and returns a pointer
*****************************************************************************/

World *create_world()
{
  World *w;

  /* allocate memory for a world */
  if((w = (World *)malloc(sizeof(World))) == NULL) {

    fatal("create_world: malloc\n");
    exit(FAILURE);
  }

  /* initialize fields */
  w->degree = 0;
  w->poly_list = NULL;

  return(w);
}
```

```
/*************************************************************************
Function name   : Composition
Purpose         : This function is used to compute the composition
                  of two given transformations
Parameters      : two two dimensional coordinate transformations
Return Type     : CONFIGURATION-the composition of the two
                  transformations q1 and q2
*************************************************************************/
CONFIGURATION composition(q1,q2)
CONFIGURATION q1;
CONFIGURATION q2;
{
   CONFIGURATION q3;
   q3.point.x = q1.point.x + q2.point.x * cos(q1.theta) -
                q2.point.y * sin(q1.theta);
   q3.point.y = q1.point.y + q2.point.x * sin(q1.theta) +
                q2.point.y * cos(q1.theta);
   q3.theta = q1.theta + q2.theta;

   return q3;
}


/*************************************************************************
Function name : Get_Circ_Transformation
Purpose       : This function is used to compute the circular transformation
Parameters    : &delta_theta , &delta_s
Return Type   : CONFIGURATION
*************************************************************************/

CONFIGURATION Get_Circular_Transformation(delta_s,delta_theta)
double delta_s,delta_theta;
{
   CONFIGURATION q;

   q.point.x = (1.0 - ((delta_theta * delta_theta)/6.0) +
               ((delta_theta*delta_theta*delta_theta*delta_theta)/120.0))
               * delta_s;
   q.point.y = (0.5 - ((delta_theta * delta_theta)/24.0) +
               ((delta_theta*delta_theta*delta_theta*delta_theta)/720.0))
```

```
                * delta_theta * delta_s;
   q.theta = delta_theta;

   return (q);
}


/*****************************************************************************
Function name : Normalize()
Purpose       : This function is used to get the value of sigma from the user
Parameters    : -
Return Type   : double
*****************************************************************************/

double Normalize(angle)
double angle;
{
   while ( angle > PI )
     {
       angle -= 2*PI;
     }
   while ( angle < -PI)
     {
       angle += 2*PI;
     }

   return angle;
}


/*********************************************************************
Function name : steer()
Purpose       : computes steering function
Parameters    : double a,double b ,double c ,
                CONFIGURATION q1,CONFIGURATION q2
Return Type   : double
*********************************************************************/

double steer(a,b,c,q1,q2)
double a,b,c;
CONFIGURATION q1,q2;
{
```

```
return (- (a * q1.kappa + b *Normalize(q1.theta - q2.theta) +
     c *(-(q1.point.x-q2.point.x)*sin(q2.theta)+
       (q1.point.y-q2.point.y)*cos(q2.theta))));

}


/*********************************************************************
Function name : Get_constants()
Purpose       : calculates the value of the constants a,b,c
Parameters    : -
Return Type   : -
*********************************************************************/

void Get_constants(sigma,a,b,c)
double *a,*b,*c,sigma;
{
  double k;
  printf("\nEnter the value of sigma please :");scanf("%lf",&sigma);

  k = 1.0/sigma;
  *a = 3 * k;
  *b = 3 * SQR(k);
  *c = QUB(k);
}


/*********************************************************************
Function name : move_next_step()
Purpose       : apply steering function , get next configuration
Parameters    : double s,double sigma ,double delta_s ,
                double delta_theta ,CONFIGURATION q1,CONFIGURATION q2)
Return Type   : CONFIGURATION
*********************************************************************/

CONFIGURATION move_next_step(a,b,c,delta_s ,q1,q2)
double delta_s,a,b,c;
CONFIGURATION q1,q2;
{
  CONFIGURATION circ;
  double lamda,delta_theta,sigma;
```

```
  lamda = - (a * q1.kappa + b *Normalize(q1.theta - q2.theta) +
     c *(-(q1.point.x-q2.point.x)*sin(q2.theta)+
            (q1.point.y-q2.point.y)*cos(q2.theta)));

  q1.kappa = q1.kappa + lamda * delta_s;
  delta_theta = q1.kappa * delta_s;
  circ = Get_Circular_Transformation(delta_s,delta_theta);

  q1.point.x = q1.point.x + circ.point.x * cos(q1.theta) -
               circ.point.y * sin(q1.theta);
  q1.point.y = q1.point.y + circ.point.x * sin(q1.theta) +
              circ.point.y * cos(q1.theta);
  q1.theta = q1.theta + circ.theta;

  return (q1);
}


/*****************************************************************************
Function :   create_polygon()
Purpose  :   create instance of a polygon
Returns  :   Polygon *
Called by:   ANYBODY
Calls    :   fatal() <utilities.c>
Comments :   this function allocates space for a polygon and returns a pointer
     to it
*****************************************************************************/

Polygon *create_polygon()
{
  Polygon *p;

  /* allocate memory for a polygon */
  if((p = (Polygon *)malloc(sizeof(Polygon))) == NULL) {

    fatal("create_polygon: malloc\n");
    exit(FAILURE);
  }

  /* initialize fields */
  p->degree = 0;
```

```
  p->vertex_list = NULL;
  p->previous = NULL;
  p->next = NULL;

  return(p);
}


/*************************************************************************
Function  :   add_vertex_to_polygon(x, y, p)
Parameters:   double x x coordinate of new vertex
              double y y coordinate of new vertex
      Polygon *p pointer to an existing polygon
Purpose   :   add a vertex to an existing polygon
Returns   :   void
Called by :   ANYBODY
Calls     :   fatal() <utilities.c>
Comments  :   adds vertex to the end of the vertex list - no way to insert
              NOTE: polygon must exist before adding vertices
*************************************************************************/

void add_vertex_to_polygon(x, y, p)
     double x, y;
     Polygon *p;
{
  int i; /* loop variable */
  Vertex
    *new_vertex, /* pointer to the new vertex */
    *current_vertex; /* pointer to the current vertex */

  /* allocate space for the new vertex */
  if((new_vertex = (Vertex *)malloc(sizeof(Vertex))) == NULL) {
    fatal("add_vertex_to_polygon: malloc\n");
    exit(FAILURE);
  }

  /* install coordinates */
  new_vertex->point.x = x;
  new_vertex->point.y = y;

  /* check if first vertex */
  if(p->degree == 0) {
```

```
    new_vertex->previous = new_vertex;
    p->vertex_list = new_vertex;
  }
  else {

    /* set up the links */
    current_vertex = p->vertex_list;
    current_vertex->previous = new_vertex;

    /* find the last vertex */
    for(i = 1; i < p->degree; i++)
      current_vertex = current_vertex->next;

    new_vertex->previous = current_vertex;
    current_vertex->next = new_vertex;

  }

  new_vertex->next = p->vertex_list;
  p->degree++;
}


/*****************************************************************************
Function  :   display_vertices_of_polygon(p)
Parameters:   Polygon *p pointer to an existing polygon
Purpose   :   display the vertices of a polygon of the existing world
Returns   :   void
Called by :   ANYBODY
Calls     :   NONE
Comments  :   -
*****************************************************************************/

void display_vertices_of_polygon(p)
Polygon *p;
{

   Vertex *current, *first;

   first=current= p->vertex_list;
   do{
     printf("\n%f %f\n",current->point.x , current->point.y);
```

```
    current = current->next;
  }while(current != first);
  return;
}


/*************************************************************************
Function  :   add_polygon_to_world(p, w)
Parameters:   Polygon *p pointer to an existing polygon
              World *w pointer to an existing world
Purpose   :   add an existing polygon to an existing world
Returns   :   void
Called by :   ANYBODY
Calls     :   NONE
Comments  :   adds polygon to end of polygon list - no way to insert polygons
*************************************************************************/

void add_polygon_to_world(p, w)
     Polygon *p;
     World *w;
{
  Polygon *current_polygon; /* pointer to current polygon */

  /* check if first polygon */
  if(w->degree == 0) {
    w->poly_list = p;
  }

  else {

    /* find the last polygon */
    current_polygon = w->poly_list;
    while(current_polygon->next != NULL)
      current_polygon = current_polygon->next;

    p->previous = current_polygon;
    current_polygon->next = p;
  }

  w->degree++;
}
```

```
/*************************************************************************
Function  :   Find_Path_Segment(v)
Parameters:   vertex
Purpose   :   computes the path segment according to a given safety distance
Returns   :   CONFIGURATION

Called by :   ANYBODY
Calls     :   NONE
Comments  :   safety distance is -50 by default(- shows ccw polygon,+ cw )
*************************************************************************/

CONFIGURATION Find_Path_Segment(v)
Vertex *v;
{
CONFIGURATION safe;
CONFIGURATION edge;

safe.point.x=0.0;
safe.point.y=-50.0;
safe.theta=0.0;
safe.kappa=0.0;
edge.point.x=v->point.x;
edge.point.y=v->point.y;
edge.theta= atan2(v->next->point.y - v ->point.y,
                          v->next->point.x - v->point.x);
edge.kappa=0.0;

return(composition(edge,safe));
}


/*****************************************
Function:   main()
Purpose :   this is the main function
*****************************************/

int main()
{
  Vertex *new_vertex;
  Polygon  *p1;
  World *w;
```

```
Vertex *v,*first;
double a,b,c,k;
double delta_s,delta_theta,sigma,lamda;
int i=0;
int N=4;
CONFIGURATION e,q[6],qo,qv;

/* make a world */
w = create_world();

/* make a polygon */
p1 = create_polygon();

/* add three vertices */
add_vertex_to_polygon(153.2, 118.0, p1);
add_vertex_to_polygon(397.2, 118.0, p1);
add_vertex_to_polygon(397.2, 182.0, p1);
add_vertex_to_polygon(153.2, 182.0, p1);

display_vertices_of_polygon(p1);

/* attach polygon to world */
add_polygon_to_world(p1, w);

OpenFile();

v=first = p1->vertex_list;
do{
   q[i] = Find_Path_Segment(v);
   v = v->next;
   i++;
   }while( v != first);

q[N]=q[0];
q[N+1]=q[1];

/* qv is the actual point we're on */
qv=q[0];/*initialization of the vehicle configuration*/

printf("\nEnter the value of delta_s please :");scanf("%lf",&delta_s);

Get_constants(sigma,&a,&b,&c);
```

```
 i=0;
 do
  {
   qv = move_next_step(a,b,c,delta_s,qv,q[i]);
   fprintf(f1,"\n%15.10f  %15.10f",qv.point.x ,qv.point.y);

  if(steer(a,b,c,qv,q[i+1])>= 0.0)
     {
     i++;/* switching lines */
     }
  }while(i<=N);

  fclose(f1);
  return 0;
}

#if 0

                                Thesis Research
                               -----------------

Filename         : star.c
Author           : Karamanlis Vasilios
Operating System : Unix
Description      : This file contains all the necessary functions to
                   track a star using the new method "neutral switching"

#endif

#include <stdio.h>
#include <math.h>

#define PI 3.14159265
#define NUM 4 *PI/5
#define SQR(x)  x*x
#define QUB(x)  x*x*x

FILE *f1;

typedef struct {
   double x;
   double y;
```

```
   double theta;
   double kappa;
}CONFIGURATION;


/*************************************************************************
Function name    : write_to_file()
Purpose          : This function is used to write the data to the
                   output file
Parameters       : two doubles
Return Type      : -
*************************************************************************/

void write_to_file(a ,b)
double a,b;
{
   fprintf(f1,"\n%10.5f %10.5f",a,b);
}


/*************************************************************************
Function name    : Composition
Purpose          : This function is used to compute the composition
                   of two given transformations
Parameters       : two two dimensional coordinate transformations
Return Type      : CONFIGURATION-the composition of the two
                   transformations q1 and q2
*************************************************************************/

CONFIGURATION
compose(first,second)
CONFIGURATION * first;
CONFIGURATION * second;
{
  CONFIGURATION third;
  double x,y, theta;
  double xx,yy,tt;

  x = second->x;
  y = second->y;
  theta = first->theta;
```

```
  xx = cos(theta) * x - sin(theta) * y +  first->x;
  yy = sin(theta) * x + cos(theta) * y +  first->y;

  tt = first->theta + second->theta;

  third.x = xx;
  third.y = yy;
  third.theta = tt;
  third.kappa = 0.0;

  return third;
}


/*****************************************************************************
Function name : Normalize()
Purpose       : This function is used to get the value of sigma from the user
Parameters    : -
Return Type   : double
*****************************************************************************/

double Normalize(angle)
double angle;
{
   while ( angle > PI )
     {
       angle -= 2*PI;
     }
   while ( angle < -PI)
     {
       angle += 2*PI;
     }

   return angle;
}


/*****************************************************************************
Function name : Get_Circ_Transformation
Purpose       : This function is used to compute the circular transformation
Parameters    : &delta_theta , &delta_s
```

```
Return Type    : CONFIGURATION
*****************************************************************************/

CONFIGURATION Get_Circular_Transformation(delta_s,delta_theta)
double delta_s,delta_theta;
{
   CONFIGURATION q;
   double delta_theta2,delta_theta4;

   delta_theta2=delta_theta*delta_theta;
   delta_theta4=delta_theta*delta_theta*delta_theta*delta_theta;

   q.x = (1.0 - ((delta_theta2)/6.0) + ((delta_theta4)/120.0)) * delta_s;
   q.y = (0.5 - ((delta_theta2)/24.0) + ((delta_theta4)/720.0))
         * delta_theta * delta_s;
   q.theta = delta_theta;

   return (q);
}


/*****************************************************************
Function name : steer()
Purpose       : computes steering function
Parameters    : double a,double b ,double c ,
                CONFIGURATION q1,CONFIGURATION q2
Return Type   : double
*****************************************************************/

double steer(a,b,c,q1,q2)
double a,b,c;
CONFIGURATION q1,q2;
{

return (- (a * q1.kappa + b *Normalize(q1.theta - q2.theta) +
     c *(-(q1.x-q2.x)*sin(q2.theta)+(q1.y-q2.y)*cos(q2.theta))));

}


/*****************************************************************
Function name : move_next_step()
```

```
Purpose        : apply steering function , get next configuration
Parameters     : double s,double sigma ,double delta_s ,
                 double delta_theta ,CONFIGURATION q1,CONFIGURATION q2)
Return Type    : CONFIGURATION
*****************************************************************/

CONFIGURATION move_next_step(a,b,c,delta_s ,q1,q2)
double delta_s,a,b,c;
CONFIGURATION q1,q2;
{
  CONFIGURATION circ;
  double lamda,delta_theta,sigma;

  lamda=steer(a,b,c,q1,q2);

  q1.kappa = q1.kappa + lamda * delta_s;
  delta_theta = q1.kappa * delta_s;
  circ = Get_Circular_Transformation(delta_s,delta_theta);
  q1=compose(q1,circ);

  return (q1);
}


/*****************************************************************
Function name : Get_constants()
Purpose        : calculates the value of the constants a,b,c
Parameters     : -
Return Type    : -
*****************************************************************/
void Get_constants(sigma,a,b,c)
double *a,*b,*c,sigma;
{
  double k;
  printf("\nEnter the value of sigma please :");scanf("%lf",&sigma);

  k = 1.0/sigma;
  *a = 3 * k;
  *b = 3 * SQR(k);
  *c = QUB(k);
}
```

```
/*****************************************************************************
Function    :   defineConfig()
Parameters  :   double x,y,theta,kappa        --The values that define a
                                               configuration
Purpose     :   To allocate nad assign a configuration
Returns     :   CONFIGURATION: a configuration
Comments    :   Was called def_configuration() in MML10
*****************************************************************************/

CONFIGURATION defineConfig(x,y,theta,kappa)
double x,y,theta,kappa;
{
  CONFIGURATION newConfig;

    newConfig.x = x;
    newConfig.y = y;
    newConfig.theta = theta;
    newConfig.kappa = kappa;

  return newConfig;
}


/*****************************************
Function :   main()
Purpose  :   this is the main function
*****************************************/

int main()
{
  int i;
  double a,b,c,k;
  double delta_s,delta_theta,sigma;

  CONFIGURATION turn,q[6],qv;

  q[0]=defineConfig(0.0,0.0,0.0,0.0);
   turn=defineConfig(300.0,0.0,NUM,0.0);

   f1=fopen("star10.dat","w");
```

```
   for(i=0; i <=5; i++)
     {
       q[i+1]=compose(q[i],turn);
       write_to_file(q[i+1].x,q[i+1].y);
     }

  /* qv is the actual point we're on */
  qv=q[0];/*initialization of the vehicle configuration*/

  printf("\nEnter the value of delta_s please :");scanf("%lf",&delta_s);

  Get_constants(sigma,&a,&b,&c);

  i=0;
  do
   {
    qv = move_next_step(a,b,c,delta_s,qv,q[i]);
    fprintf(f1,"\n%15.10f  %15.10f",qv.x ,qv.y);

   if(steer(a,b,c,qv,q[i+1])>= 0.0)
      {
      i++;/* switching lines */
      }
   }while(i<=5);

   fclose(f1);
   return (0);
}
```

# LIST OF REFERENCES


[1] Kanayama, Y., "A Path Tracking Method with Neutral Switching" Technical Report of the Department of Computer Science, Naval Postgraduate School, Monterey, California, 1997.

[2] Kanayama, Y.,Fahroo, F., "A Circle Tracking Method for Nonholonomic Vehicles" Technical Report of the Department of Computer Science, Naval Postgraduate School, Monterey, California, 1997.

[3] Kanayama, Y., " Introduction to Theoretical Robotics," *Lecture Notes of the Advanced Robotics Course*, Deptartment of Computer Science, Naval Postgraduate School, Winter Quarter 1997.

[4] Wahdan, M., "New Motion Planning and Real-Time Localization Methods Using Proximity For Autonomous Mobile Robots," Dissertation, Naval Postgraduate School, Monterey, California, September 1996.

[5] Preparata, P., and Shamos, I., "Computational geometry:An Introduc tion."Springer-Verlag, 1985.

[6] Iyengar, S., and Elfes, A., "Autonomous Mobile Robots:Perception, Mapping and Navigation." IEEE Computer Society Press Tutorial.

[7] Pobil, P.,and Serna, M., "Spatial Representation and Motion Planning" Springer-Verlag.

[8] Schwartz, J., Sharir, M., and Hopcroft, J., "Planning, Geometry and Complexity of Robot Motion" Ablex Publishing Corporation, Norwood, New Jersey.

[9] Latombe, J., "Robot Motion Planning" Kluwer Academic Publishers.

[10] Nakamuka, S., "Applied Numerical Methods in C" Prentice Hall

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   8725 John J. Kingman Road., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library — 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Hellenic Navy General Stuff(GEN)/B2 — 1
   Stratopedo Papagou
   Xolargos, Greece

4. Chairman, Code CS — 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5101

5. Professor Yutaka Kanayama, Code CS/Ka — 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5101

6. Professor Nelson Ludlow, Code CS/Ld — 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5101

7. Maj. Khaled Morsy, Code CS/Ph — 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5101

8. LIEUT. Vasilios Karamanlis (Greece) — 3
   Grevenon Emilianou 8//Ippodromio Thessaloniki
   54621//Greece